CARM – 2012 R Workshops
Centre for Applications in Natural Resource Mathematics

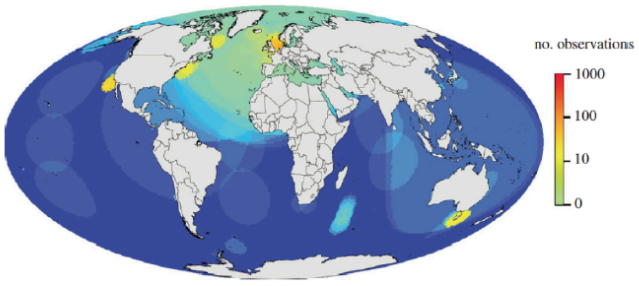
Introductory R Workshop

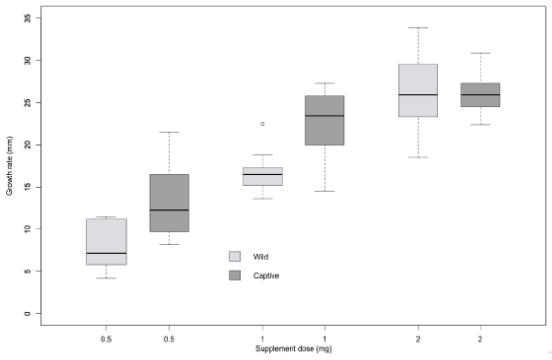
19-21 Nov 2012

Anthony J. Richardson (UQ, CSIRO, a.richardson@maths.uq.edu.au)

David S. Schoeman (University of the Sunshine Coast, dschoema@usc.edu.au)

Chris J. Brown (UQ, christo.j.brown@gmail.com)





THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

University of the Sunshine Coast
Queensland, Australia

Program

Day	Topic	Time	Presenter
1. Basics	Setup	8:30-9:00	All
	Introduction to R, RStudio, data importing and manipulation	9:00-10:30	Ant
	Morning tea	10:30-10:50	
	Summary statistics	10:50-12:00	Dave
	Simple graphics I	11:30-12:30	Ant
	Lunch	12:30-13:30	
	Simple graphics II	12:30-13:00	Ant
	Simple statistics	14:00-15:30	Dave
	Questions Homework: Simple statistics	15:30-16:00	Dave
2. Modelling	Homework solution (voluntary): Simple statistics	8:30-9:00	Dave
	Simple linear models	9:00-10:00	Ant
	ANOVA & post-hoc tests I	10:00-10:30	Dave
	Morning tea	10:30-10:50	
	ANOVA & post-hoc tests II	10:50-11:45	Dave
	Linear models & model selection I	11:45-12:30	Ant
	Lunch	*12:30-13:30*	
	Linear models & model selection II	13:30-14:15	Chris
	GLM (binomial/Poisson errors) I	14:15-14:40	Ant
	Afternoon tea	14:40-15:00	
	GLM (binomial/Poisson errors) II	15:00-15:30	Ant
	Questions Homework: Linear models	15:30-16:00	Ant
3. Multivariate and programming	Homework solution (voluntary): Linear models	8:30-9:00	Ant
	Multivariate statistics I	9:00-10:30	Dave
	Morning tea	10:30-10:50	
	Multivariate statistics II	10:50-12:30	Chris
	Lunch	*12:30-13:30*	
	Programming basics	13:30-14:30	Dave
	Afternoon tea	14:30-14:50	
	Programming intermediate: functions	14:50-15:45	Chris
	Closing comments/discussion/Survey Monkey	15:45-16:00	All

DAY 1

Introduction to R

Who we are

We are not hardcore statisticians, but are biologists, who have an interest in statistics, and use R frequently. Our aim in this three-day introductory workshop is to guide you through the basics of using R for analysis of biological data. This is not a comprehensive course, but is necessarily selective. Our emphasis will be on analysing data in R, rather than focusing on the statistical theory. If you feel overwhelmed at any time during the workshop, then that is totally natural with learning a new and powerful program. Remember that you have the notes to go through the exercises later at your own convenience. This course should get you started with R, and if you are already a user, it will hopefully show you some new ways of doing things.

Why learn R?

As a scientist, we increasingly need to analyse and manipulate datasets. Our datasets are growing in size and our analyses are becoming more sophisticated. There are many statistical packages on the market one can use, but R is fast becoming the global standard, for a number of reasons.

1. It is free
2. It is powerful, flexible and robust
3. It contains advanced statistical routines not yet available in other packages
4. It has state-of-the-art graphics capabilities
5. It is popular, and so has a massive user community who help each other
6. The user community continually extends the functionality

But you will have to learn to program…

The big difference between R and other statistical packages is that it is not a menu-driven 'point and click' package and never will be. R requires you to write your own computer code to tell it exactly what you want done. Although this means there is a learning curve, there are many advantages:

1. To write new programs, you can modify existing ones or those of others, saving you considerable time
2. You have a record of your statistical analyses and thus can re-run your previous analysis exactly at any time in the future, even if you cannot remember what you did
3. It is more flexible in being able to manipulate data and graphics than menu-driven packages
4. You will develop, and then improve your programming, a valuable skill
5. You will improve your statistical knowledge
6. For transparency and repeatability, journals are starting to request programming code that underpins published analyses
7. Programming is challenging and fun!

Getting started with RStudio

We will use RStudio in this workshop. RStudio is a free front-end to R (*i.e.*, R is working in the background). It makes working with R more productive, straightforward and organised, especially when beginning. RStudio has four main windows: Source editor, Console, Workspace, and Plots.

Console window

This is where you can type commands that execute immediately. Throughout the notes, we will represent code for you to execute in R following a ">" symbol and in a different font. Note that you do not enter ">" in R, as this is the automatic prompt at the start of a line in the Console window.

Entering code is easy. For example, we can use R as a calculator by typing (and pressing Enter after each line):

```
> 6+3
> 5 * 8
> 2^ 4
```

Note that spaces are optional around simple calculations.

We can also use the assignment operator `<-` (read as 'gets') to assign any calculation to a variable so we can access it later:

```
> a <- 2
> b<-7
> a + b
```

Spaces are also optional around assignment operators. I use single spaces extensively in my programs to make them more readable. An important question here is – is R case sensitive? Is `A` the same as `a`? Figure out a way to check for yourself.

We can also assign a vector by using the combine (`c`) function:

```
> apples <- c(5.3, 3.8, 4.5)
```

Finally, there are many inbuilt functions in R, including the mean and standard deviation:

```
> mean(apples)
> sd(apples)
```

RStudio supports the automatic completion of code using the Tab key. For example, type the following and then the Tab key:

```
> app
```

The code completion feature also provides inline help for functions whenever possible. For example, type the following and press the Tab key:

```
> read
```

Other ways to get help in R and RStudio from the Console include:

```
> ?mean
```

OR

```
> help(mean)
```

The RStudio console also supports the ability to recall previous commands using the Up and Down arrow keys (a complete history of executed code is available in the History tab, which is described below).

Source editor

The Source editor helps with your programming. Let us open a simple program file from the menu:

Go File/Open File and choose the file HelloWorld.r

Note the extension .r for R program files. These are simply standard text files with a .r extension. They can be created in any text editor and saved with a .r extension, but the Source editor provides syntax highlighting, code completion, and smart indentation. You can see the different colour code for numbers and there is also highlighting to help you count brackets (put your cursor insertion point before a bracket and push the right arrow and you will see its partner bracket highlighted). We can execute R code directly from the source editor. Try the following (for Windows machines; for Macs replace Ctrl with Cmd):

Execute a single line (Run icon or Ctrl+Enter). Note cursor can be anywhere on the line
Execute multiple lines (Highlight lines with cursor, the use Run icon or Ctrl+Enter)
Execute whole program (Source icon or Ctrl+Shift+Enter)

Now, try changing the NumOfIterations in HelloWorld.r, run it, and see what happens.

Now let us save the program in the Source Editor by clicking on the file symbol (note that when the file has changed since the last time it was saved, it has a * beside the .r extension in the program name tab).

Workspace, History windows

The Workspace shows your variables and data. You can see the values for variables with a single value and for those that are longer, R tells you their class and you can click on them and their values will appear in the Source Editor. Also in the Workspace is the History tab, where you can see all the commands for the session. You can also send any of these commands to the Console (*i.e.*, run them) or to the Source editor (*i.e.*, copy the code).

Files, Plots, Packages, Help windows

The last window has a number of different tabs. The Plot tab is where graphics appears. There is also the Help tab, where the help appears when you ask for it from the Console. You can search the R documentation by selecting the Help tab and typing your request in the top right text box. You can also install packages via the Packages tab. Note that the list is not all the packages that R has – they are all available from the CRAN (**C**omprehensive **R A**rchive **N**etwork) website. A useful feature of RStudio is that you can also Check for Updates of the packages that you have installed, as there are regular updates for many of them.

For those able to access the internet, let us install the package "lme4". Under the Packages Tab, click the Install Packages button that links with the CRAN website. Type in the Package name in the text box (note that capitals are important). Also note that it will install dependencies (*i.e.,* it will install other R packages that are needed to run the target package). It installs the packages on your hard drive. You can then load the package via ticking the checkbox for that package. (Note that the difference between a library and a package in R is that a library is the directory in which you can find R packages). RStudio automatically runs the R code needed to install the package, so if you want to include these in one of your programs just copy the text.

Question: Why is it best practice to include packages you use in your R program explicitly?

Configuring windows in RStudio

Note that you cannot rearrange windows in RStudio by dragging them, but if you prefer an arrangement that differs from the default, you can make such changes via the Pane Layout tab via the Tools/Options (RStudio/Preferences – for Mac) menu.

Importing data in R

We will see how easy it is to import data into R now. R will read in many types of data including spreadsheets, text files, binary files, and files from other statistical packages. R is also a good platform for manipulating your data.

Importing data can actually take longer than the statistical analysis itself! An important aspect to remember is that to be able to analyse your data, it needs to be in an appropriate, yet strict, format. Generally, within each column (variables), the format needs to be precisely the same and is commonly of the following types. A continuous numeric variable (*e.g.*, fish length (say in m): 0.133, 0.145); a factor categorical variable (*e.g.*, Month: Jan, Feb or 1, 2, …, 12); an ordered factor (*e.g.*, three levels of nutrient enrichment: low, medium or high); or a nominal variable (*e.g.*, algal colour: red, green, brown). You can use other more specific formats such as dates, though, and general formats such as any text.

Before we import some data, we need to set a working directory. This is where R will look for any imported files, and where R will save any files it writes.

In the Files tab, navigate to the RIntroductoryWorkshop directory. Then under More, select the small down pointing triangle and select Set As Working Directory. This means that whenever you read or write a file then it will always be working in that directory. To set the working directory directly from the Console or within a program, copy the code that RStudio runs. Below is for my laptop and it will be different for you):

```
> setwd("/Users/ric325/Documents/CARM - Centre for Applications in
Natural Resource Mathematics/R_IntroductoryWorkshop/")
```

Note that if you copy from a path in Windows the slashes will be the wrong way around. We can check that the working directory is set to what we want:

```
> getwd()
```

Question: Why is it best practice to include the working directly explicitly in your R program?

Now we have the working directory set correctly, R will know where to look for our import file. The function `read.table` is the most convenient way to read in statistical data. To find out what it does, we will go to its help entry:

```
> ?read.table
```

All R help items are in the same format. A short Description (of what it does), Usage, Arguments (the different inputs it requires), Details (of what it does), Value (what it returns), Examples. The Arguments are the lifeblood of any function, as this is how the information is provided to R. You do not need to specify all arguments, as some have appropriate default values for your requirements and some will not be needed for your particular example.

There are many arguments so that you can use to customise your import, but most important are:

1. file: data file to be read

2. header: the variable (column or field) names in the header argument. It is important to name variables appropriately. It is safest to have no spaces, no funny characters, no function names (*e.g.*, mean) among the variable names
3. sep: the separator between fields. The most common separator is the comma, as that is unlikely to appear in any of the fields in English-speaking countries. Such files are known as CSV (comma separated values) files
4. quote: By default, character strings can be quoted by either single '"' or double ''' quotes and usually do not need to be changed when exporting data as .csv from Excel.

We are going to import in the file BeachBirds.csv. In RStudio, go to the Workspace tab and select the Import Dataset button and then the From Text File… option. This provides a GUI for data import. Select the file BeachBirds.csv and peruse the Input File and the Data Frame. There is a header row that provides the variable names. There is a mix of categorical (only a few different levels – *e.g.*, Species, Sex and Site) and numeric variables (continuous variables – flush.dist and land.dist). There are also missing data represented by the code NA. Select the Import button and the file will be imported. It saves the data frame with the name of the file.

Note that we can also import in the dataset using code in the Source editor (the code that R uses is given in the Console when we imported in the data using RStudio). First, let's clear the variables in memory by selecting Clear All button from the Workspace tab (using the broom icon). Now let's start writing our first program by clicking on the NewDocumentButton in the top left and selecting R Script. This gives us an unnamed file. Best to save it first of all so we do not lose what we do. File/Save As/ and it should come up in the Working Directory and type in "BeachBirds". It will automatically add a ".r" extension.

It is recommended to start a program with some basic information for you to refer back to later. Start with a comment line (a # in R) that tells you the name of the program, something about the program, who created it, and the date it was created. In the source editor enter:

```
> # Beachbirds.R. Reads in and manipulates bird data. <YourName>
<CurrentDate>
```

The next line is probably where you would set your working directory. Even though it is already set, it is best to include it in your program. Why?:

```
> setwd("c:/Users/ric325/Documents/CARM - Centre for Applications in
Natural Resource Mathematics/R_IntroductoryWorkshop/")
```

and then:

```
> dat <- read.table("BeachBirds.csv", header= TRUE, sep = ",")
```

Or even simpler for csv files:

```
> dat <- read.csv("BeachBirds.csv", header= TRUE)
```

`read.csv` actually calls `read.table` with appropriate defaults. You can choose either one in your program.

Now look at the variable `dat` in the Workspace by clicking on the variable name. Check that it has imported in correctly. If we look at the help for `read.csv`, we can see that the Value (output) is a data frame. This is the fundamental data structure that R uses.

A comment about missing data. Having missing data represented by a blank in a .csv file is usually OK because the data elements are separated by commas, but if you import a space delimited format then it can have difficulty with missing values being blank. Once imported in R, the missing data should be coded as NA (Not Available), for both text and numeric variables. However, it is safer to replace missing data (blanks) in your spreadsheet with NA, the missing data code in R. If you have blanks for missing data in Excel, just before importing the data into R you can highlight the area of the spreadsheet that includes all the cells you need to fill with NA. Do a Edit/Replace… and leave the "Find what:" textbox blank and under the "Replace with:" textbox enter NA, the missing value code. Once imported into R, the NA values will be recognised as missing data.

Also note that there are packages in R to read in Excel spreadsheets (*e.g.*, `xlsReadWrite`), but remember there are formulae, graphs, macros and multiple worksheets in spreadsheets that can compromise import. We recommend exporting data deliberately to csv files (which are also commonly used as import to other programs).

Now let's do some simple data manipulation. You will need to do this in almost every program you write. If we want to refer to a variable, we specify the data frame, a "`$`" sign (meaning within an object), and then the variable name. At the Console type:

```
> dat$Sex
> dat$flush.dist
```

Now let us use this terminology to specify certain rows. Note that within a dataframe, the rows are numbered from 1 to number of rows, and the columns from 1 to number of columns. So if we want to select only the rows for Site 2 then we use the `which()` function. Again at the Console type:

```
> dat$Site == 2
```

What does this give you? Note that here we are not assigning `dat$Site` to 2 (*i.e.*, we are not using `<-` or `=`), but using `==` which queries `dat$Site` for when it equals 2.

To find the rows in `dat` that correspond to Site == 2, we write:

```
> which(dat$Site == 2)
```

Other operators we can use include:

Operator	Meaning	Example
>	greater than	x > 3
<	less than	x < 3
>=	greater than or equal to	x >= 3
<=	less than or equal to	x <= 3
!=	not equal to	x! = 3

Logical operator	Meaning	Example
&	AND. returns TRUE if both statements on either side of the & are TRUE	(x > 3) & (y != 5)
|	OR. The pipe symbol "|". TRUE if either statement on either side of the | is TRUE	(x < 10) | (x > 20)

Now, let's determine the rows that include both Site 2 and Site 4 (i.e., Site has the values of 2 or 4). At the Console type:

```
> which(dat$Site == 2 | dat$Site == 4)
```

Note if we said:

```
> which(dat$Site == 2 & dat$Site == 4)
```

We get nothing returned. Why? Note that we use square brackets when specifying indices.

Task: In your program you have started, create a new data frame containing only female Plovers with `land.dist` values >80? Hint: First determine the row indexes required.

Finally, you will be left with many variables and data frames after working through these examples. Note that in RStudio when you quit it saves the Workspace and so will retain the variables in memory when you start RStudio again. It is good to clear the variables in memory. Type the following code to get a list of variables in memory:

```
> ls()
```

Note that because you have written code, you can just re-run it to generate the variables again. You can remove them using the function `rm()`:

```
> rm(list = ls())
```

The parameter list provides a list of variables to be deleted (you could concatenate all existing variable names together in quotes using `c()`, but more easily the function `ls()` gives all the variable names in memory.

Tip: It is particularly useful to use the `rm(list = ls())` command at the start of new programs, so the program starts with no predefined variables. It must be at the start though before you define any variables.

Summary Statistics

The data

In the last section we imported the file BeachBirds.xlsx into R and assigned it to a data frame named `dat`. These data reflect results of an experiment on beaches designed to measure the influence of off-road vehicles (ORVs) on shorebirds. We visited five different beaches (Sites), and at each site, drove along the shoreline in an ORV. As we drove along, we identified birds in the distance, and drove at them until they took flight. We recorded the species and sex of the bird, the distance from the bird at which it took flight (flush.dist), as well as the distance the bird flew before settling again (land.dist). In instances where sex could not be determined, or where birds flew out of sight before landing, we marked observations "NA".

Checking data

Once the data are in R, the next thing we may be interested in is checking that there are no glaring errors.

I usually call up the first few lines of the data frame using the function `head()`. Try it yourself by typing:

```
> head(dat)
```

This lists the first six lines of each of the variables in the data frame as a table. You can similarly retrieve the last six lines of the data frame by an identical call to the function `tail()`. Of course, this works better when you have fewer than ten or so variables; for larger data sets, things can get a little messy. If you want more or fewer rows in your head or tail, tell R how many rows it is you want by adding this information to your function call. Try typing:

```
> head(dat, n = 3)
```

If you're interested in checking the names of the variables listed in the data frame you've imported, type:

```
> names(dat)
```

You can also check the structure of your data by typing:

```
> str(dat)
```

This function lists the variables in your data frame by name, tells you what sorts of data are contained in each variable (*e.g.*, continuous number, discrete factor) and provides an indication of the actual contents of each.

Summary of all variables in a data frame

Once we're happy that the data have imported correctly, and that we know what the variables are called and what sorts of data they contain, we can start to dig a little deeper. Try typing:

```
> summary(dat)
```

The output is quite informative. It tabulates variables, and for each provides summary statistics. For continuous variables, the name, minimum, maximum, first, second (median) and third quartiles, and the mean are provided. For factors (discrete variables), a list of the levels of the factor is given. In either case, the last line of the table indicates how many NAs are contained in the variable.

Summary statistics by variable

This is all very convenient, but we may want to ask R specifically for just the mean of a particular variable. In this case, we simply need to tell R which summary statistic we are interested in, and to specify the variable to work on. Try typing:

```
> mean(dat$flush.dist)
```

Note that `$`specifies the element of the data frame (`dat`) that is called `flush.dist`. This convention is worth remembering, as it provides easy access to named variables.

There is another way of specifying a variable within a data frame, using the `with()` function:

```
> with(dat, mean(flush.dist))
```

Of course, the mean isn't the only summary statistic that R knows about. Try `max()`, `min()`, `median()`, `range()`, `sd()` and `var()`. Do they return the values you expected? Now try:

```
> mean(dat$land.dist)
```

The answer probably ISN'T what you would expect. Why not? Well, sometimes, you need to tell R how you want it to deal with circumstances. In this case, you have NAs in the named variable, and R takes the cautious approach of giving you the answer of NA, meaning that there are missing values here. This is not very useful, but as the programmer, you can tell R to respond differently, and it will. Simply append this argument to your function call, and you will get a different response. Try typing:

```
> mean(dat$land.dist, na.rm= TRUE)
```

The `na.rm` option tells R to REMOVE (or more correctly "strip") NAs from the data string before calculating the mean. It now returns the correct answer.

More complex calculations

This is all very useful, but let's say you want to calculate the standard error of the mean for a variable, rather than just the standard deviation. How can this be done?

The trick is to remember that R is a calculator, so we can use it to do complex maths. The formula for standard error is:

$$standard\ error = \sqrt{\frac{variance}{n}}$$

We know that the variance is given by `var()`, so all we need to do is figure out how to get *n* and then combine the two values to get the answer we want. The simple way to determine the number of elements in a variable is a call to the function `length()`. Try typing:

```
> length(dat$flush.dist)
```

Bearing this in mind, we can calculate standard error as follows:

```
> sqrt(var(dat$flush.dist)/length(dat$flush.dist))
```

Alternatively, we could use symbolic notation by assigning objects to contain the values for the numerator and the denominator. Try typing:

```
> numerator <- var(dat$flush.dist)
> denominator <- length(dat$flush.dist)
```

Here we are creating variables called numerator and denominator and populating them with numeric values. We can then calculate the standard error and assign it to an object by typing:

```
> stderr <- sqrt(numerator/denominator)
```

To get a printout of the standard error, you can simply type:

```
> stderr
```

Alternatively, you could also navigate to the *Workspace* window in RStudio and click on the variable called `stderr`. While you are there, try clicking on `dat`. What happens?

Tip: At ALL times, remember that R acts to alert you to the presence of NAs; this gives you the opportunity to express explicitly how they should be dealt with.

When calculating the mean, we specified that R should strip the NAs, using the argument `na.rm`. In the example above, we didn't have NAs in the variable of interest. What happens if we DO?

Unfortunately, the call to the function `length()` has no arguments telling R how to treat NAs; instead, they are simply treated as elements of the variable and are therefore counted. The easiest way to resolve this problem is to strip out NAs in advance of any calculations, perhaps by specifying a new variable. Try typing:

```
> length(dat$land.dist)
```

then

```
> length(na.omit(dat$land.dist))
```

You will notice that the function `na.omit()` removes NAs. Using this new information, calculate the mean landing distance and the corresponding standard error.

Now, these sorts of summary statistics are fine for continuous variables, but we may want something a little different for discrete variables. Try typing:

```
> table(dat$Species)
```

This returns a table with a column corresponding to each unique level of the discrete variable. In the first line is the value of the level (in this case, the names of the bird species we observed in our experiment); in the second line is the frequency of occurrence of that level in the variable (the number of birds of each particular species encountered).

Of course, this is easily extended to two-way tables. Try typing:

```
> table(dat$Species, dat$Site)
```

Again, we could alternatively use:

```
> with(dat, table(Species, Site))
```

This lists the number of observations for each species at each site, with sites as columns, and species as rows.

Question: Can you figure out how to get a table reflecting the overall sex ratio of each species? Lay the table out so that each species has its own column, with a row of counts for each sex. Note that there are NAs where the sex was indeterminate. How did R deal with these?

So far, we have worked with tables cross-classifying two variables. What happens when we have three? Try typing:

```
> threeway <- table(dat$Species, dat$Sex, dat$Site)
```

This seems useful, but still a little messy. Luckily for us, the good people who write R were real human beings, so they provide neat little tricks for reformatting things. Try typing:

```
> ftable(threeway)
```

Summarising data from a table or matrix

A table is really just a matrix (*i.e.*, a collection of data arranged by row and column). Often, we will need to compute statistics from such data structures. Let's go back to a table we constructed before, and assign it a name; type:

```
> sp_by_site <- table(dat$Species, dat$Site)
```

Confirm that it looks right by typing:

```
> sp_by_site
```

Now, what if we wanted to check that the rows in this table sum correctly? We already know that we can output the number of observations from each species by typing:

```
> table(dat$Species)
```

But are these totals the same as in the species-by-site table? Try typing:

```
> apply(sp_by_site, MARGIN = 1, FUN = sum)
```

`apply()` applies functions over array margins. Here the argument `MARGIN` tells R whether you want to apply the function (`FUN`) to the rows or columns; we set `MARGIN` to 1, so we got the sums of values in each row. Rerun the code after changing the `MARGIN` to 2. What do you get?

Of course, in other contexts, we may be interested in calculating the mean or standard deviation, or whatever, and we can easily do this by simply changing the function.

More complex summaries by group

Very often in science, we're going to be interested in calculating summary statistics by group. For example, we may be interested in calculating the mean flushing distance by species. Or maybe we're interested in calculating the mean by species and site. Try typing:

```
> tapply(dat$flush.dist, INDEX = list(dat$Site, dat$Species), FUN = mean)
```

or equivalently:

```
> with(dat, tapply(flush.dist, INDEX = list(Site, Species), FUN = mean)
```

NOTE: The more times you have to write out the data frame name in your code, the more useful the function `with()` becomes. In the remaining text we will use the `$` convention to specify variables just for clarity, but you may well benefit from using `with()` instead.

The `tapply()` function is a special case of `apply()`, and you can see that it refers to an `INDEX` (categorical identifiers for groups) rather than to the `MARGINS` of a matrix. The output is a data frame (which can be very useful, as we'll see later in the course).

Although `tapply()` is pretty useful, the output isn't formatted for ready use as a table. Luckily there are other ways of doing the same sort of thing. Try typing:

```
> aggregate(dat$flush.dist, by = list(dat$Species, dat$Site), FUN = mean)
```

This provides an alternative arrangement of the data we generated using `tapply()`.

Task: Now that you have the skills, try calculating the mean distance that birds fly once disturbed, by species, sex and site. Remember that there are NAs in this data set. What are our options for dealing with these?

Saving data in a useful format

A major advantage of R over many other statistics packages is that you can generate exactly the same answers time and time again, by simply re-running saved code. However, there are times when you will want to output data to a file that can be read by a spreadsheet program like Excel. I find that the simplest general format is csv (comma-separated values). This format is easily read by Excel, and also by many other software programs. To output a csv is simple. Try typing:

```
> write.csv(sp_by_site, file = "BirdTable.csv", row.names = TRUE)
```

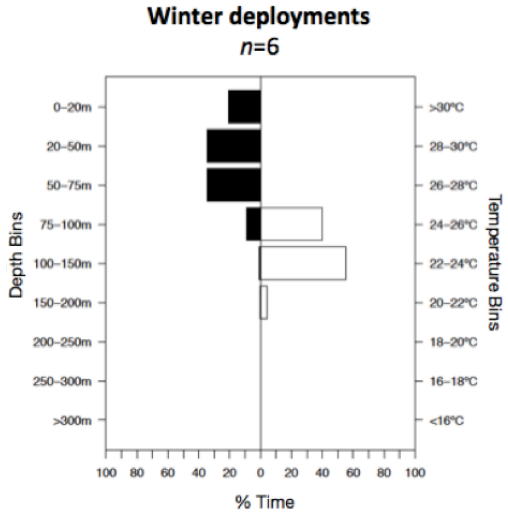
The first argument is simply the name of an object, in this case our table of counts by species and site (other sorts of data are available, so play around to see what can be done). The second argument is the name of the file you want to write to. This file will always be written to your working directory, unless otherwise specified (we'll chat about this later). The last argument simply tells R to add a column of values specifying row names (in this case, the names of the bird species). Of course, if you don't want row names (which might be the case if you were writing the whole data frame to a file), simply replace the "TRUE" with "FALSE". The resultant file can be imported to Excel using the File/Import/CSV File menu chain (or by right-clicking the file name and selecting Open With Excel).

Simple graphics

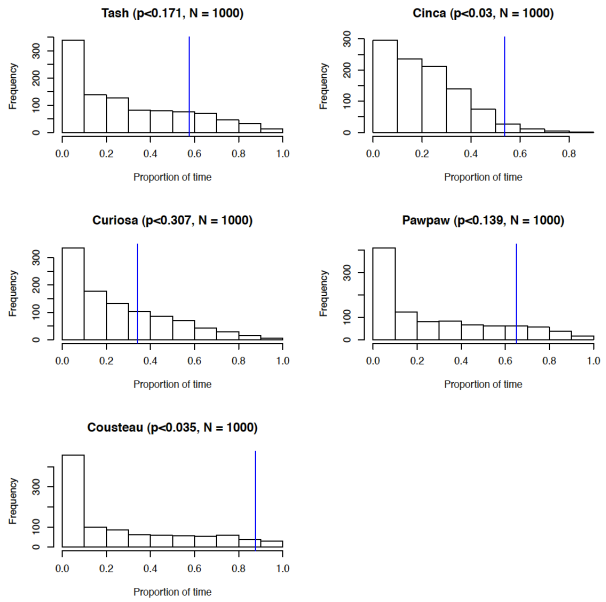
R has powerful and flexible graphics capabilities. In this workshop we will only cover traditional graphics (in the graphics package automatically loaded in R). There is the ability for extensive customization in traditional graphics, so it will cover many of the graphs that you will want to produce. There are also many R packages that have plotting functions that extend the traditional graphics available (we will see some of these when we do some mapping later in the workshop). Let us start by quickly seeing some capabilities of traditional R graphics and then we will learn some general principles.

Standard plots

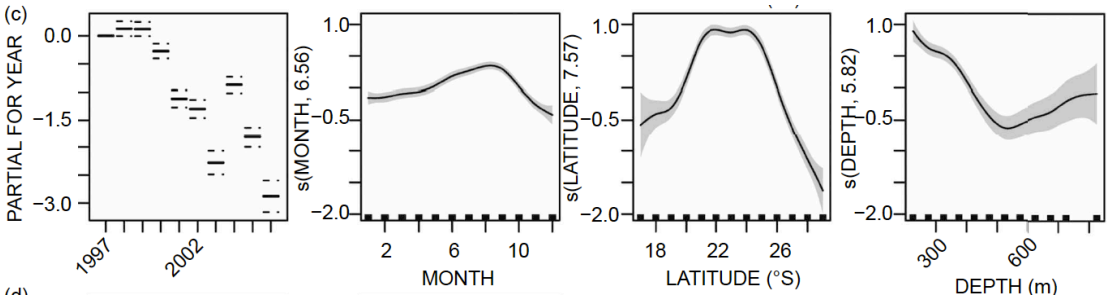
To give you a flavour of graphics capabilities in R, here is a range of graphics we have produced:



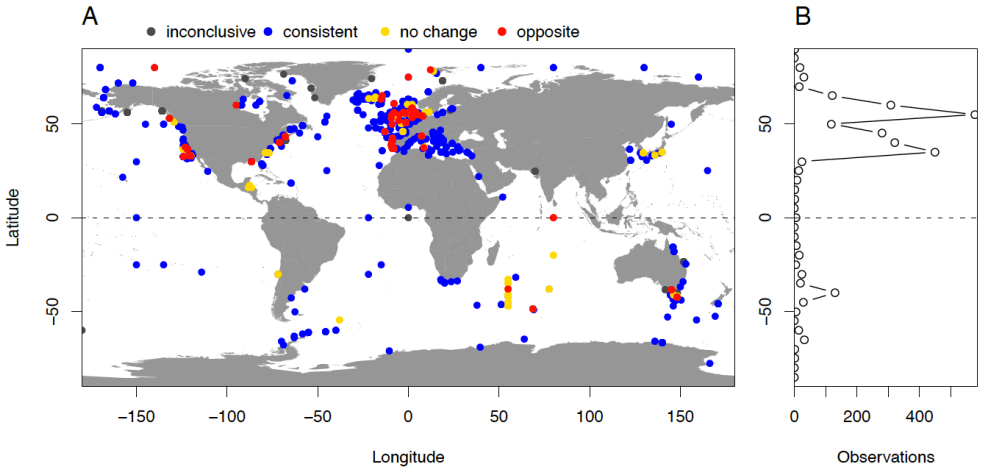
Customising simple plots: Plotting time at depth and time at temperature for satellite tags on manta rays.


Tash (p<0.171, N = 1000)
Cinca (p<0.03, N = 1000)
Curiosa (p<0.307, N = 1000)
Pawpaw (p<0.139, N = 1000)
Cousteau (p<0.035, N = 1000)
Frequency
Proportion of time

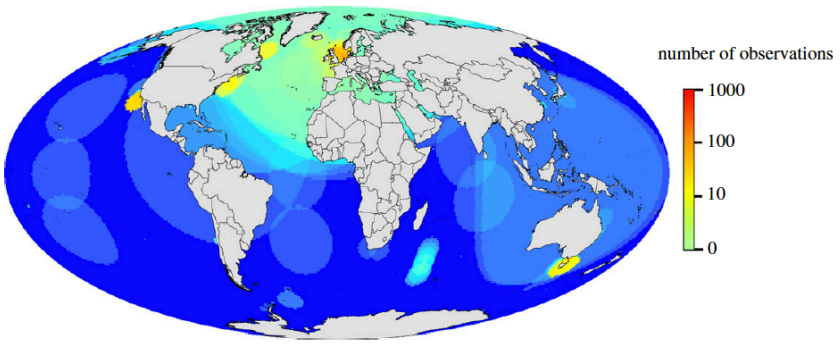
Multiple panels: The proportion of time that different manta rays spend within the Capricorn eddy (blue line) compared with 1000 'modelled' manta rays that travelled the same distance over the same time but in random directions.



Statistical output: The abundance of jellyfish in Namibia. Shown is the output from a generalised additive model. The response is the transformed proportion of jellyfish occurrence in fish trawls, and predictors are Year, Month, Latitude and Depth.



Mapping: A. Marine biological time series that are consistent, opposite, no change or inconsistent with climate change. B. Number of time series by latitude.



Mapping: A density plot of marine biological time series more than 19 years in length used to assess climate change impacts.

Producing graphics

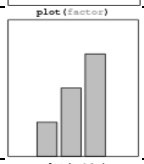
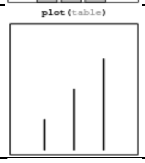
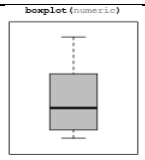
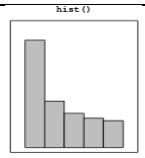
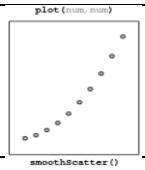
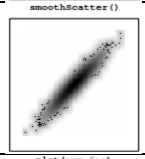
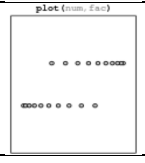
These graphs are just variations of traditional graphics. There are many types of graphs that can be drawn, and these are summarised in Table 1. A good way to categorise the different types of graphs is by number of variables they use: 1, 2 or multiple variables. Have a read through some of the help entries for these

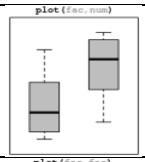
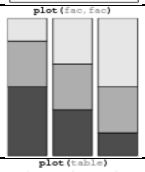
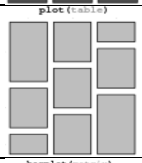
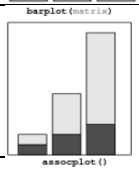
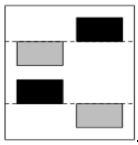
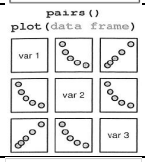
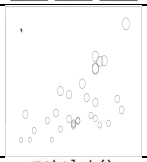
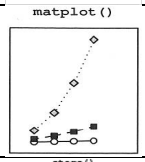
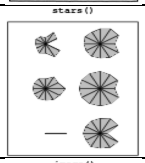
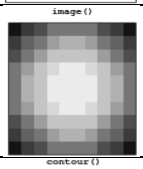
graphs. Note that there is an Examples section at the bottom of most help menus for R functions. Examples can be run by typing (for example, for boxplots):

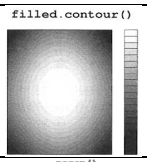
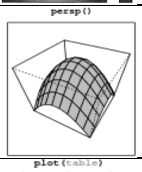
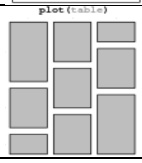
```
> example(boxplot)
```

and pressing return to scroll through the graphs. This is a good way to see what the graphs look like.

Summary of useful standard plots (there are many others!). Modified from R Graphics by Paul Murrell.

#Variables	Function	Data	Description	Example
1	`plot()`	Numeric	Scatterplot. Assumes sequential x value (as line plot in Excel)	
	`plot()` or `barplot()`	Factor	Bar plot. `plot()` is frequency histogram of factor. `barplot()` when bar heights in dataset	
	`plot()`	1-D table	Bar plot. Used after the table command	
	`pie()`	Numeric	Pie chart. When areas of pie slices in dataset	
	`boxplot()`	Numeric	Box-and-whisker plot. Solid line (median), box (upper and lower quartiles), whiskers (range excluding outliers), points (outliers)	
	`hist()`	Numeric	Generates frequency histogram from raw data. Dependent upon bin size, so kernel density plots can be better way of viewing distribution of a variable plot(density(x))	
2	`plot()`	Numeric, numeric	Scatterplot. Plots x vs y	
	`smoothScatter()`	Numeric, numeric	Smoothed colour density representation of scatterplot	
	`plot()`	Numeric, factor	Strip chart. 1-D scatter plot of data. Good alternative for boxplots with small samples sizes	

	`plot()`	Factor, numeric	Box-and-whisker plot	
	`plot()`	Factor, factor	Spine plot showing relative frequency of observations in each group	
	`plot()`	Table	Mosaic plot of contingency table (*i.e.,* shows relative frequency of observations in each group). Calls `mosaic.plot()`	
	`barplot()`	Matrix	Stacked or side-by-side bar plot. Height of bars given in dataset	
	`assocplot()`	2-D table	Association plot shows number of observations in each group	
Multiple	`plot()` or `pairs()`	Data frame or matrix	Scatterplot matrix of all variables against each other. Factors are treated as integers	
	`symbols()`	Numeric, numeric, numeric	Symbol scatterplot. Useful for showing values on a 3rd dimension in 2-D	
	`matplot()`	Matrix	Scatterplot of multiple x variables and multiple y variables	
	`stars()`	Matrix	Star plots. Good for multivariate data (say <20 dimensions) in a number of groups. With one location, also draws 'radar' plot	
	`image()`	Numeric, numeric, numeric	*e.g.*, satellite images, bathymetry maps are in a flat 2-D array or as X and Y for grid line locations for data Z	
	`contour()`	Numeric, numeric, numeric	*e.g.*, satellite images, bathymetry maps are in a flat 2-D array or as X and Y for grid line locations for data Z	

	`filled.contour()`	Numeric, numeric, numeric	*e.g.*, satellite images, bathymetry maps are in a flat 2-D array or as X and Y for grid line locations for data Z	
	`persp()`	Numeric, numeric, numeric	*e.g.*, satellite images, bathymetry maps are in a flat 2-D array or as X and Y for grid line locations for data Z	
	`plot()`	N-D table	Mosaic plot of contingency table (*i.e.*, shows relative frequency of observations in each group). Calls `mosaic.plot()`	

Plotting basics

The plot() function

The simplest and most important command in R graphics is the `plot()` function. You can customise many graph features (*e.g.*, colours, axes, titles) through specifying graphic options in the plot call.

Let's see how we use it by generating some random numbers to plot:

```
> y1 <- runif(30, 0, 100) # 30 random numbers between 0 and 100
> plot(y1)
```

Careful: Occasionally there is a problem with plotting in RStudio; it sometimes returns an error that margins are too small. Simply resize the Plot window in RStudio to fix it.

Note here that we used a hash (#) to tell R not to run any of the text on that line to right of the symbol. This is the standard way of commenting R code; it is VERY good practice to comment in detail so that you can understand later what you have done.

Specifying plot types

The y-values here are plotted against an assumed sequential index for x-values. By default, points are plotted. It is easy to change the type of plot by specifying the type parameter. The following are the possible plot types:

"p" for points,
"l" for lines,
"b" for both lines and points,
"c" for the lines part alone of "b",
"o" for both 'overplotted',
"h" for 'histogram' like vertical lines,
"s" for stair steps,
"S" for other steps
"n" for no plotting of points (useful to set up a space to add graphics at a later stage)

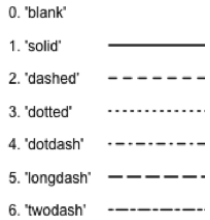
For example:

```
> plot(y1, type = 'l') # line plot
```

Note that either single or double quotes can be used for text variables in R. Try some of these other plot types.

Specifying lines: types, thickness and colour

The parameter `lty` specifies the line type and can be an integer or keywords:



Note that when we specify a line plot, a solid line is automatically drawn (*i.e.*, the default value for `lty` is '`solid`'). Let us produce a line plot with a dashed line:

```
> plot(y1, type = 'l', lty = 'dashed')
```

Try using the different line types in the list above.

Now, let us change the line thickness. The parameter `lwd` specifies the line width and is a positive value defaulting to 1 (*e.g.*, 0.1 is very narrow and 10 is very thick):

```
> plot(y1, type = 'l', lty = 'dashed', lwd = 3)
```

Try changing `lwd`.

Let us now draw a red line. To make it red, we can use one of the pre-specified colours in R:

```
> plot(y1, type= 'l', lty = 'dashed', lwd = 3, col = 'red')
```

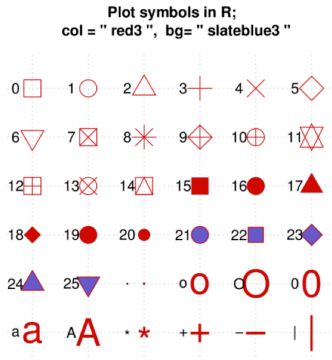
R has a very extensive list of pre-defined colours available:

```
> colours()
```

Try changing the colour of the line. The default line colour is black.

Specifying points: symbol, line widths, size and colour

We can change the type of symbol used for the points by changing the parameter `pch` (stands for 'plotting character'). The `pch` parameter can either be a number or text. For each pair below, the number or text you enter is on the left and the plotted symbol is on the right:



The default value is 1 (circles). Now let us specify points as upward pointing triangles:

```
> plot(y1, pch = 2)
```

Note that any text symbol can be plotted if it is specified within quotes. Try some different symbols and text. We can change the colour of the symbol using the `col` parameter again:

```
> plot(y1, pch = 2, col = 'blue')
```

For unfilled symbols, we can change the line thickness using the `lwd` parameter

```
> plot(y1, pch = 2, col = 'blue', lwd = 3)
```

We can also change the symbol size using the character expansion parameter `cex`. This is a number giving the amount by which plotting symbols should be magnified relative to the default. Let us make the symbols larger:

```
> plot(y1, pch = 2, col = 'blue', lwd = 3, cex = 1.5)
```

As you can see, R is pretty flexible with modifying the attributes of plotting symbols.

Specifying multiple data series on a plot: using lines and points

R graphics follows a "painter's model", where graphics output is drawn sequentially, so current output overlays existing output (until the next `plot()`). Let us plot our initial random numbers as blue circles and a second series as red squares. We will draw the second series from the normal distribution, with means incrementing from 35 to 65 and with a standard deviation of 10. We will then join them with lines:

```
> plot(y1, col = 'blue')
> y2 <- rnorm(30, 35:65, 10) # 30 random numbers from normal
distribution
> points(y2, pch = 0, col = 'red')
> lines(y2, pch = 0, col = 'red')
```

Specifying lines: using lines and abline

Let us plot our random numbers again:

```
> plot(y1, col = 'blue')
```

Now let us add a horizontal dashed line at 50 to represent the mean of the 1st series of random numbers. We will use the lines command and draw a line from position (0,50) to (30, 50):

```
> lines(x = c(0, 30), y = c(50, 50), lwd = 2, lty = 'dashed', col =
'blue')
```

Now let us plot our 2nd series of random numbers and plot a line through the `points` y2. The function `abline()` takes as parameters the intercept and slope. This would commonly be done by fitting a model first (see Regression section), but for now we will assume that the line of best fit has an intercept of 35 and a slope of 1.

```
> points(y2, pch=0, col = 'red')
> abline(a=35, b=1, col='red', lty='dotted') # a = intercept; b = slope
```

Note that `abline()` can also be used to draw horizontal or vertical lines, as follows:

```
> points(y2, pch = 0, col = 'red')
> abline(h = 10) # A horizontal line at 10 on the y-axis.
> abline(v = 12) # A vertical line at 12 on the y-axis.
```

Specifying titles, axis labels and free text: colours and size

We can specify a title by providing a text string to the parameters main:

```
> plot(y1, main = 'Time series')
```

Axis labels can be added by providing text strings to `xlab` and `ylab`:

```
> plot(y1, main = 'Time series', xlab = 'Index values', ylab = 'Random
y-values')
```

We can change the colour for parameters `main` and `lab`:

```
> plot(y1, main = 'Time series', xlab = 'Index values', ylab = 'Random
y-values', col.main = 'blue', col.lab = 'blue')
```

Different font styles can be specified with the parameter `font.main` and `font.lab` (possible values include 1 = plain (default), 2 = bold, 3 = italic, 4 = bold and italic):

```
> plot(y1, main = 'Time series', xlab = 'Index values', ylab = 'Random
y-values', col.main = 'blue', col.lab = 'blue', font.main = 4, font.lab
= 2)
```

You can also change the size of fonts (and points) by using the `cex.main` and `cex.lab`:

```
> plot(y1, main = 'Time series', xlab = 'Index values', ylab = 'Random
y-values', col.main = 'blue', col.lab = 'blue', font.main = 4, font.lab
= 2, cex.main = 2, cex.lab = 0.75)
```

We can also specify that the symbols (`col`), axes (`fg`), and axes tick marks (`col.axis`) are blue:

```
> plot(y1, main = 'Time series', xlab = 'Index values', ylab = 'Random
y-values', col.main = 'blue', col.lab = 'blue', font.main = 4, font.lab
= 2, cex.main = 2, cex.lab = 0.75, col = 'blue', col.axis = 'blue', fg =
'blue')
```

Finally, you can position a text label anywhere on the plot with the command text:

```
> text(x = 2, y = 53, labels = 'Feeling blue…', col = 'blue')
```

Specifying default graphics parameters – par()

Sometimes you want to use the same parameters for several plots. Default graphics parameters can be set with the `par()` function. Some parameters can only be set with `par()`. To see the current `par()` settings:

```
> par()
```

Scroll through the list. You can see that there are many parameters that can be changed. As any parameter values set with the par command are in effect for the rest of the session (*i.e.*, all subsequent graphs) or until you change them again, it is good practice to first store a copy of the current `par` settings so you can reset them later to their initial values:

```
> oldpar <- par()      # stores a copy of current settings in oldpar
```

Now let us make the default plotting symbol green filled triangles, and plot multiple figures, using the `mfrow` parameter ("**m**ultiple **f**igures in a **row** layout"). It requires a vector of the form `c(number of rows, number of columns)`:

```
> par(mfrow = c(1,2), pch = 17, col = 'green') # 1 x 2 plots
> plot(y1)
> plot(y1,y2)
```

Each sequential call to plot will put that graph into the next window.

Task: Now let's do some example plots using the bird data again (BeachBirds.xlsx). Import the data into a data frame. Using the summary table of standard plots above, make four different plots, each with two variables: a scatterplot, a box and whisker plot, a strip chart and a spine plot

Plot secondary y-axis

To illustrate the flexibility in R graphics, we'll do an example where we plot two series of data, with different y scales, on the same plot but on different axes. This is a common problem and builds on commands we already know. Try the following (with bird data in the data frame `dat`):

```
> par(mar = c(5, 4, 4, 5) + 0.1) # set plot margins wider for 2nd axis
> plot(dat$flush.dist, dat$land.dist, xlim = c(0, 25), xlab = "Flush
distance ", ylab = "Land distance")
> par(new = TRUE) # subsequent calls to plot() will plot over what you
already have
> plot(1:24, runif(24), axes = FALSE, pch = 19, xlim = c(0, 25), ylim =
c(0,1), xlab = "", ylab = "")
> axis(4) # adds the tick marks and labels
> mtext("Random numbers", side = 4,line = 3) # plot text in the margin
for side = 4 (2nd y-axis) and on 3rd line of margin
```

The trick is to set `xlim` explicitly in both plots if you have different x-values in each.

Saving a graph

Now, let us save the graph as a pdf. In RStudio, we can use the Export button under the Plots tab and save as pdf. We can also write code to export graphics in many different formats (see below). We need to ensure we switch off the graphics device after doing this so the file is written:

```
> pdf('TestPlot.pdf') # Open the printing device
> …all the code for the plot goes in here…
> dev.off() # Close the printing device
```

Function	Output to
pdf("mygraph.pdf")	pdf file
win.metafile("mygraph.wmf")	windows metafile
png("mygraph.png")	png file
jpeg("mygraph.jpg")	jpeg file
bmp("mygraph.bmp")	bmp file
postscript("mygraph.ps")	postscript file

Reset `par()`

Finally, let us reset the par() parameters:

```
> par(oldpar) # reset original par settings
```

Note that resetting the graphics parameters sometimes returns warnings. Warnings are just that – R is telling you that your code has executed, but that some parts might have returned unexpected results. In this case, the warning is nothing to worry about, but in general, look closely at any warnings that pop up.

Simple statistics

A simple one-tailed, one-sample t-test

Numerical and graphical summaries of our data are valuable, but to get published, we generally have to test hypotheses. And R does these pretty well. Let's open a new dataset similar to the one we used previously: BeachBirds2.xlsx. Again, assign these data to a data frame called `dat`.

You know how to have a look at the data structure to make sure it all makes sense, so I won't go through this again here. Just bear in mind that it's good practice to check.

Let's suppose that beach-driving legislation suggests that to prevent putting birds to flight, ORV drivers should not drive within 10 m of them. Do our data support the efficacy of this legislation?

To set a null hypothesis requires a tiny bit of thinking. Given the context provided (drivers need to stay at least 10 m away from birds), our research question is:

> Is the mean flushing distance of birds greater than 10 m?

Our hypotheses are then:

> H_0 (null): Mean flushing distance of birds ≥ 10 m
> H_A (alternate): Mean flushing distance of birds < 10 m

We can test this null hypothesis using a simple function in R. Try typing:

```
> t.test(dat$flush.dist, mu = 10, alternative = "less")
```

Here, we are specifying that we want to run a one-sample t-test (we are supplying only one variable), that our hypothesised mean (`mu`) is 10 m, and that our `alternative` hypothesis contains a less-than sign (remember that the null hypothesis always contains some form of equality). Being an obedient computer program, R returns all of the information we would expect:

1. A test statistic (t-value)
2. The degrees of freedom of the test (df)
3. The associated p-value

Results strongly support rejecting the null hypothesis ($p < 2.2e\text{-}16$), and this is confirmed by the mean provided at the end of the data summary of 8.19 m. This strongly suggests that remaining at least 10 m away from birds should, on average, avoid disturbing them so severely that they take flight, so the regulation is doing its job.

In this case, we used a one-tailed test, because our null hypothesis wasn't one of equality. Moving to a two-tailed test is as simple as specifying the "alternative" option as "two.sided", or just leaving it out altogether.

A two-sample t-test

In ecology, we often encounter questions that require a two-sample, rather than a one-sample approach. For example, we may be interested to know whether flight responses of gulls differ from those of oystercatchers. In this case our null and alternative hypotheses might look something like this:

H_0: Mean distance to landing of disturbed gulls = that of oystercatchers
H_A: Mean distance to landing of disturbed gulls ≠ that of oystercatchers

This poses a question: what data formats are suitable for this sort of test, and how do we get the data into that format? Although this may seem daunting, it is not, because R has simple and powerful ways of subsetting data. Try typing:

```
> gulldat <- subset(dat, Species == "Gull")
```

This selects a subset of the data frame `dat` that contains only data pertaining to gulls and saves it as a data frame. Now create a second subset named `oycdat` containing the subset of `dat` that pertains to oystercatchers.

Once we have these new data frames, the flight distances for gulls will be called `gulldat$land.dist` and those for oystercatchers will be called `oycdat$land.dist`.

Try calling these variables to see what they look like (alternatively, select them in RStudio's *Workspace*). Calculate their respective means. Would we expect these to differ significantly?

Let's run the formal test and see. Try typing:

```
> t.test(gulldat$land.dist, oycdat$land.dist)
```

In this instance, we simply supply R with two columns of data and ask it to compute the statistics for the test, which it does easily. The output is very similar to that for a one-sample t-test, and you will notice from the means at the end that the test automatically discards NAs. The result confirms that mean flight distances differ significantly between these two bird species.

The non-parametric equivalent of the two-sample t-test

Let's suppose now that we're interested in whether there is the same number of gulls as stints on the beaches. Because we drove at each bird we saw, we have a measure of bird abundance at each site. The only problem is that counts tend to be distributed according to a Poisson rather than a Normal distribution (all data are non-negative integers and the variance of the sample tends to be the same as the mean). If data are non-normal, one approach is to use a non-parametric analysis, so this is what we'll do here (in reality, though, t-tests and other parametric tests are reasonably robust to violations of this assumption). We will see later in the section on Generalised Linear Models how to explicitly cope with Poisson error structures.

Again, we first need to gather the data into an appropriate format. Remember that the `table()` function will make a frequency table for us, so try typing:

```
> table(dat$Site, dat$Species)
```

Indexing data by row and column number

Notice that the data we are interested in are in the first and fifth columns of the table. Although we'll deal with this in detail later in the course, it's worth pointing out the matrix-type data structures in R have an indexing convention. Perhaps an analogy with Excel is the place to start. In Excel, every cell is named by column, then row, so that the cell in the upper left-hand corner of a spreadsheet is A1 (Column A, Row 1), etc. R uses a very similar convention, but reverses this order and uses numbers to identify both columns and rows. So, for example, the number of gulls at Site 1 is in cell 1,1, whereas the number of stints at Site 1 is in cell 1,4 (remember that R uses Row, Column).

Let's use this idea to access some of the data in the table. First, to simplify things, let's assign the table to an object:

```
> counts <- table(dat$Site, dat$Species)
```

Now access the number of gulls at Site 1. Type:

```
> counts[1,1]
```

Notice that we access this data by telling R the name of the object we want to interrogate, and then which element or elements (identifying that we want an element by enclosing its position within square brackets). Now that we know this, get the number of gulls at Site 5, and the number of stints at Site 3.

Taking this further, we could get the number of plovers at each site using the following code:

```
> counts[c(1, 2, 3, 4, 5), 3]
```

Or, because you can generate a sequence of numbers from 1 to 5 by typing 1:5...:

```
> counts[c(1:5), 3]
```

Or, even simpler:

```
> counts[1:5, 3]
```

Or, better still, you can simply omit the row identifier altogether, in which case R assumes that you want ALL rows in the matrix, which we want here:

```
> counts[, 3]
```

Use this convention to output number of birds per species at Site 2.

Given this information, we can now try a non-parametric test of the hypotheses:

H_0: Abundance of gulls = abundance of stints
H_A: Abundance of gulls ≠ abundance of stints

To do this, try typing:

```
> wilcox.test(counts[,1], counts[,4])
```

Note that the counts per Site of gulls is held in column 1 of the frequency table, whilst counts of stints are in column 4. We could, of course, have assigned these values to objects and used the object names in the test, but the way we have done it is quicker.

The output again provides a test statistic and a p-value (which helps us to conclude that the species indeed differ in abundance), but also issues a warning about calculating exact p-values in the case of ties. Bear in mind that R does NOT ALWAYS warn you when assumptions are violated.

Task: Run a two-sample t-test using the same data. Which of the two tests (parametric or non-parametric) is more likely to detect a difference (here we get a clue from the size of the p-value)?

Simple correlations

So, simple significance tests are easy to do in R. What about relationships among data series? Import the Excel data called OceanProd.xlsx, and again, assign these data to a data frame called `dat`.

These data describe results of an oceanographic survey of 50 stations selected at random from Australia's EEZ. The variable SST describes sea-surface temperature (ºC), PP is primary production (g C m^{-2} yr^{-1}), SP is zooplankton abundance (numbers per standard net haul), and FP is commercial fish catch (tonnes per year). We are interested in assessing relationships among components of the food web.

Because we cannot tell *a priori* whether the food web is bottom-up or top-down forced (prey limited or predator limited), there are no real predictor or response variables, so correlation is appropriate. Let's start by asking whether there is a relationship between primary production and zooplankton abundance. If we assume that the data comply with the usual assumptions of parametric analysis, we can use Pearson correlation. Try typing:

```
> cor(dat$PP, dat$SP)
```

This returns the Pearson correlation coefficient without any test statistics. If we want to formally test the null hypothesis that the correlation coefficient is zero (*i.e.*, that these variables are unrelated), we type:

```
> cor.test(dat$PP, dat$SP)
```

This returns a full suite of test statistics, demonstrating that we can reject the null hypothesis ($p << 0.05$), indicating a strong and significant relationship between primary and zooplankton production in Australian waters.

Task: Use a correlation test to determine the relationship between zooplankton production and fish catch.

If we had reason to believe that our data did not conform to the assumptions of parametric analyses, we could have simply altered the default "`method`" of the correlation test from Pearson to either Kendall or Spearman, by simply specifying this option. For example, try typing:

```
>  cor.test(dat$PP, dat$SP, method = "spearman")
```

Plots of pair-wise correlations:

Often in exploratory analyses, we are interested in simply "eyeballing" the relationships among many variables simultaneously. R very usefully offers a simple method to do this. Try typing:

```
> pairs(dat)
```

This produces a matrix of plots, with the variable names on the diagonal, and corresponding scatter plots reflected either side. Actual correlations can be accessed by typing:

```
> cor(dat)
```

There are more sophisticated things you can do with correlations, but we'll leave it there for now.

Homework: Day 1

Background

As a beach ecologist, I am interested in the ecosystem services that coastal sediments might provide. One of these is the ability of resident microbes to oxidise ammonium into nitrites and nitrates. Because I suspect that urban centres discharge greater amounts of ammonium into the groundwater than do rural areas, it is possible that the accelerating urbanisation of coastlines may impact this ecosystem service. To test the idea, I design a simple pilot study, based on the beaches of the Sunshine Coast.

Ten beaches were selected, and each was assigned an index of urbanisation (small values indicate relatively low urban pressure, whereas high values suggest heavy urbanisation). At each beach, I selected five points at random along the driftline (the line of flotsam and jetsam left by the highest recent high tide) and called these sampling sites. On the outgoing tide, I inserted a pair of piezometers (a piezometer is a narrow, temporary well - think of it as a drinking straw inserted into the sand to the depth of the water table) at each sampling site on each beach. One of the paired piezometers was inserted at the driftline and the other at the effluent line (the line at which groundwater seeps from the beach, forming a glassy layer of saturated sand). At low tide, I extracted a water sample from each piezometer pair, and recorded the concentration of ammonium (in μg.L-1). All data are available in the Excel spreadsheet called Ammonium.xlsx. The beach name is provided in the column named Beach, the urbanisation index is in Urban, the sample site is in Site, the shore level is in Level, and the ammonium concentration is in NH4. Where samples were contaminated or lost, this is marked in the dataset as NA.

Use these data to address the following research questions:

1. What is the mean, standard deviation, standard error, and range of ammonium concentrations at the driftline on beaches of the Sunshine Coast?
2. Plot the mean ammonium concentration at the driftline by beach.
3. Which beaches have an urbanisation index of at least 0.5?
4. What is the mean ammonium concentration at the effluent line of Mooloolaba Beach?
5. Provide a box-and-whisker plot showing the difference in ammonium concentrations between the driftline and effluent line on Noosa Beach.
6. Overall, is there a decrease in ammonium concentrations between the driftline and the effluent line on beaches of the Sunshine Coast?

R Introductory Workshop

DAY 2

Simple linear models

Many of the problems we encountered in the previous section on simple statistics (e.g. t-tests and even the example where we had count data) can be tackled using a linear modelling. The approach is to fit a model to the data. Model fitting in R is part of its real strength. Once we know how to specify a simple model, it is not much more difficult to specify a complex one. We will start by understanding simple models before moving onto more complex ones.

Why fit models?

Our objective is to determine the values of parameters in a model that best explain the data. The model is always fitted to the data, not the other way round. We are guided by three key principles:

1. Our mechanistic understanding of the problem in our variable choice and their functional forms,
2. The principle of parsimony that states that our model should be as simple as possible,
3. The adequacy of the model in describing a substantial fraction of variation in the observed data.

R will not produce its own models; it will fit models that you ask it to. The onus is on you.

There are many statistical models that you can fit in R, including `lm`, `aov`, `glm`, `gam`, `lmer`, `nls`, `nlme`, `loess`, `tree`, and many more. We will only deal with some of these in this workshop, but thankfully they all have the same structure for specifying formulae, so building models in R is fairly generic.

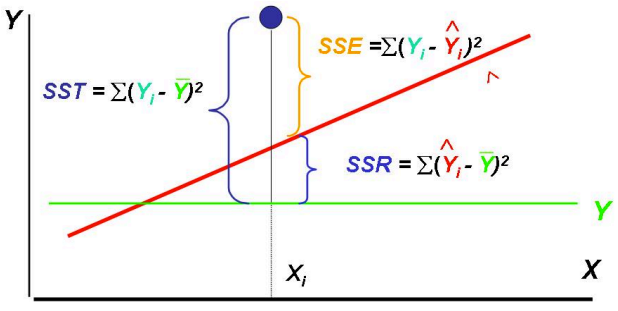
Sums of squares

The basis for fitting elementary models is the concept of minimising the sums of squares. It is easiest to think of it in terms of simple linear regression, but the logic is the same for cateogorical explanatory variables. Sums of squares are defined as:

Y_i = observations

$\overline{Y}$ = overall mean

$\hat{Y}_i$ = estimated y-values



SST = Sums of Squares Total

SSR = Sums of Squares Regression

SSE = Sums of Squares Error

In a linear model, we seek to minimize the error (residual Sums of Squares). For simple least squares regression, it is effectively rotating the fitted line until the error sums of squares are minimised. This gives the slope and intercept. Linear models use Sums of Squares to calculate parameter values and assess their significance. For example, when we assess the significance of the slope parameter, we use the F-test, which is the ratio of two variances to see if they are significantly different, is SSR/SSE.

Question: Why do we use squared deviations rather than just deviations (*i.e.,* without the squared sign)?

Model formulae in R

The first part of fitting a model is specifying the formula. The structure of a statistical model in R is:

response variable ~ explanatory variable(s)

Where the tilde symbol (~) is read as "is modelled as a function of"

Thus a simple linear regression of two continuous variables **y** and **x** is:

y ~ x

and a 1-way ANOVA with Season having 4 levels (*i.e.,* summer, autumn, etc.) would be:

y ~ Season

Thus, R knows whether you are doing a regression or an ANOVA by the class of the explanatory variables

Remember: Whether variables are categorical or continuous needs to be specified before a model is fit. If the variable x was a categorical variable (factor) representing months and was specified in words (*e.g.,* Jan, Feb, …, Dec), then on import x will be assumed to be a factor. However, if you specify months as numbers from 1 to 12, then on import R will assume this is a continuous variable. To let R know that you want months to be considered as categorical, you would use:

```
> factor(x)
```

The right hand side of the formula shows the number and identity of explanatory variables, interactions between explanatory variables if applicable, and any non-linearity in the explanatory variables.

Variables can also be transformed in the model, so we can do a log transformation of the response:

log10(y+1) ~ x

There are many built-in maths functions in R that you can include in formulae (and use elsewhere, too) Some examples are:

```
> sqrt()
> log()                          # natural log
> log10()                        # base-10 logs
> sin()                          # and all the other trig
> abs()                          # absolute value
```

```
> round()                          # and floor(), ceiling()
```

So to square-root transform the response, you could either create a new variable:

```
> y2 <- sqrt(y)
> y2 ~ x
```

or simply reflect the transformation in the formula:

```
> sqrt(y) ~ x
```

While a model formula resembles a mathematical formula, the symbols on the right side of the ~ are interpreted differently (note `y` is a continuous response and `x`, `x1` and `x2` are continuous explanatory variables and `A` and `B` are categorical explanatory variables):

Symbol	Explanation	Examples
`+`	inclusion of an explanatory variable	`y ~ x + A`
`-`	deletion of an explanatory variable	`y ~ . – A`
`.`	The model as it stands	`y ~ . + x`
`:`	An interaction. `A:B` is interaction between `A` and `B`. Interaction occurs when the effect of `A` and `B` together on the response is not equal to the sum of their individual effects. We need to know value of `A` to know effect of `B` on response and *vice versa*	`y ~ A + B + A:B` `y ~ A + x + A:x`
`*`	Inclusion of explanatory variables and their interaction (*i.e.*, the full factorial model). Common for categorical variables	`y ~ A * B`, which is the same as `y ~ A + B + A:B`
`/`	Nesting of an explanatory variable. `A/B` is `B` nested within `A`	`y ~ A/B`, which is the same as `y ~ A + A:B`
`poly`	Polynomial regression. Useful for incorporating non-linearities	`poly(x,3)` is a 3rd order polynomial

`s()`,
`lo()`,
`nb()`
	Smoothers for non-linear terms	`y ~ s(x1) + s(x2)`
		`y ~ lo(x1) + lo(x2)`
		`y ~ nb(x1, df = 2) + nb(x2, df = 3)`
		where `x1` and `x2` are continuous

`I`	Include "as is"	`y ~ I((x-10)*(x>10))` Breakpoint regression with constant y values for `x<10` and a slope for `x>10`
`cbind`	**C**olumn **bind** two vectors together	`y~cbind((x-10)*(x<10),(x-10)*(x>10))` Break point regression with 2 different slopes either side of `x=10`

The lm model

The simplest models are fit with `lm()`, which stands for **l**inear **m**odel. Let's investigate a simple example where growth rate of kelp is negatively related to the concentration of arsenic. The data are contained in the file KelpGrowth.csv. Read in the data file, check the variable names and the class of the variables.

To run the `lm()`, we tell R the response and the explanatory (predictor) variable, and where they are:

```
> model1 <- lm(Growth ~ Toxin, data = Data)
```

It is as easy as that. This is the general regression model:

$$y = a + bx$$

The `data` argument for `lm()` specifies the data frame where the variables are.

Results of `lm()` are stored in the object `model1`. Let us see what the results are:

```
> summary(model1)
```

It starts by giving you a summary of the model call. There is then a summary of the residuals to give you an idea if the model has fitted OK. We will see more model diagnostics soon. There is a summary table of information concerning the coefficients in the model. This gives the Estimate (value of the coefficient), its Standard error, its t-value, and its significance. So we can see that Toxin has a significant (at $p<0.01$) negative effect on kelp growth rate. The model has 7 degrees of freedom (total number of observations of 9 minus 2 degrees of freedom, one for each parameter estimated (intercept and slope)). Toxin explains a large amount of the variation in zooplankton growth ($r^2=0.79$). Remember that:

SST = SSR + SSE

Note that SSR is the same as SSModel (*i.e.*, the sums of squares for the whole model if you have multiple continuous and categorical variables)

r^2 = SSR/SST

The adjusted r^2 is also given, which is the r^2 adjusted for the number of parameters in the model. The more parameters the better the model fit – and the adjusted r^2 is penalised for more parameters in the model.

Note that by default the intercept (*a* in the general model) is included in the model and so does not need to be specified. Note that if you want to fit a model with only an intercept (so no slope), it would be:

```
> model2 <- lm(Growth ~ 1, data = Data)
```

Check: Now look at the value for the intercept. Compare it with the mean for Growth. What is happening here?

We can also look at the ANOVA table, which shows the sums of squares.

```
> anova(model1, test = "F")
```

Here we ask R to produce an ANOVA table and to test the significance of each parameter in the model using an F-test. The ANOVA table shows that the significance of the slope from the F-test in the ANOVA table is the same as that for the t-test from the `summary()` call.

Check: Is the r^2 calculated from the sums of squares given in the ANOVA table the same as that from the `summary()` call? You can calculate this from the Sum of Squares by taking the ratio of the Sums of Squared for the model (here Toxin) to the total Sums of Squares.

To calculate the confidence interval for each parameter, we use `confint()` and set the confidence level.

```
> confint(model1, level=0.95)
```

Now let's plot the data and the line of best fit. We can use the `abline` function:

```
> plot(Data$toxin, Data$growth)
> abline(model1)
```

Does this work? If not, why not? When you pass a simple linear regression model (y~x) to `abline`, it automatically uses the intercept and slope, which we can find using the coefficients function:

```
> coefficients(model1)
```

Predicted values

To obtain predicted values from the model, we can use the `predict()` command:

```
> predict(model1)
```

This provides the predicted values at the values of the explanatory variables in the original data frame. To get predictions for any other values for the explanatory variables, we can provide these within a data frame (here Toxin values from 1.0, 1.5, 2.0, 2.5, …, 14.5, 15):

```
> predict(model1, newdata = data.frame(Toxin = seq(1,15,0.5)))
```

Note that we use the function `seq()` here. This specifies that we want R to generate a sequence running from 1 to 15 in increments of 0.5. This is useful in many contexts, so try to remember it. The function `data.frame()` converts the sequence of numbers into a data frame, with the variable named Toxin.

Model diagnostics

Before accepting the results, we need to see how well the model has fit. The major assumptions in linear modelling are normality and homogeneity of variance. It is common to use a qualitative approach in model diagnostics to assess a model, rather than perform significance tests for normality and homogeneity of variance, as linear models are often robust to mild departures from the assumptions (and tests of the assumptions generally have low power). We can assess model residuals with:

```
> residuals(model1)
```

However, the simplest assessment is using four model-checking plots:

```
> par(mfrow = c(2,2))
> plot(model1)
```

The top left graph shows residuals on the y-axis against fitted values on the x-axis. We can use this plot to assess the assumption of homogeneity of variance; if it is met, residuals should look like the 'stars at night' – the points scattered with little pattern, as it does here. The worst situation is if the residuals increase as fitted values increase, as this implies the variance increases with the mean. If the pattern was a horseshoe shape, then it could indicate that the model was mis-specified (*e.g.*, assuming a linear term when a curvilinear one might be more appropriate). In the top right is the `qqnorm` plot, which should be a straight line if errors are normally distributed. The example here looks fine. If it were S- or banana-shaped, then errors would not be normally distributed and a different error structure might be needed. The bottom-left plot is the same as the top left, but on a different scale. The bottom-right plot shows Cook's Distance plotted against leverage. This helps us identify points with the biggest influence (outliers). For each point, a regression with and without the point is performed, and the squared difference between slopes is Cook's Distance.

ANOVA and post-hoc comparisons

The general linear model, as exemplified by the simple linear regression in the last section, has a specific application in ecology where the predictor variable is categorical (discrete) rather than continuous. We refer to this commonly as Analysis of Variance or ANOVA. Running ANOVA in R is relatively straightforward, and follows from the methods we used in the previous section.

To illustrate, please load up the data on our beach-driving experiment from BeachBirds2.xlsx and assign the data frame to the data frame `dat`.

Let's assume that we want to know whether different species of birds all flush at the same rate or not. In this case, our null hypotheses look something like this:

H_0: Flushing distance of gulls = flushing distance of oystercatchers = flushing distance of plovers = flushing distance of stints
H_A: Flushing distance of gulls ≠ flushing distance of oystercatchers ≠ flushing distance of plovers ≠ flushing distance of stints

So our null hypothesis is that ALL flushing distances are identical.

We can test the model using a format identical to the one we used in the section before. Try typing:

```
> mod1 <- lm(flush.dist ~ Species, data = dat)
```

As before, to extract the useful information, we must ask for a model `summary()`:

```
> summary(mod1)
```

The output looks a little tricky. First, there seem to be estimates for each species except gulls. Second, some of these estimates are negative, so they can't refer to flushing distances.

In fact, the `(Intercept)` refers to the flushing distance of gulls, and the associated p-value tells you that it is significantly different from zero ($p << 0.05$). This is known as the reference level of the discrete predictor variable. By contrast, the estimates for the remaining species refer to flushing distance of that species less the intercept (which, as we have explained already, is the flushing distance for gulls). In this sense, the flushing distance for oystercatchers is:

8.17 m + 0.37 m = 8.54 m

Similarly, the flushing distance for plovers is:

8.17 m - 0.12 m = 8.05 m

And so forth. Note that p-values for these estimates indicate that the amount to be added to the intercept is no different from zero. In other words, the mean flushing distances for all other birds do not differ significantly from those of gulls. This is confirmed by the overall p-value, provided right at the end of the summary: $p = 0.7821$.

Hopefully you're thinking that it would be interesting to set some other species as the baseline for comparison . Although this would make no difference here, there may be many cases where such an operation would be very useful; for example, where you have a control group against which you wish to test all other (experimental) groups. Resetting the baseline in this way can be achieved using the function

`relevel()`. For example, this function can be used as follows to modify the data frame so that Stints become the baseline:

```
> dat$Species <- relevel(dat$Species, ref = "Stint")
```

Now that you've done this, try refitting the model and inspecting the summary.

You might be thinking that this format of output is overly complicated, and perhaps it is, but it comes back into play later in the course when we're talking about generalised rather than general linear models. R recognises this complication and provides a more familiar output by simply calling the function `aov()` rather than `lm()`. To illustrate, try typing:

```
> mod2 <- aov(flush.dist ~ Species, data = dat)
> summary(mod2)
```

This looks more like the ANOVA table we know, with degrees of freedom (df), sums of squares (Sum Sq), mean squares (Mean Sq), the F statistic, and an associated p-value. Note that the degrees of freedom, F and p are identical to those from the end of the summary of `mod1`, showing that this is exactly the same model, just fit a different way. In fact,

```
> summary(aov(flush.dist ~ Species, data = dat))
```

is equivalent to

```
> anova(lm(flush.dist ~ Species, data = dat))
```

Try it for yourself to see.

Note that if you look at the structure of the data using:

```
> str(dat)
```

Species is listed as a Factor. In other words, R has recognised that it is a discrete variable, and has allowed analyses to operate accordingly. If we were to set hypotheses about Site, however, we may have some problems, because R identifies this as a variable containing integers, so analyses may become confused. To avoid this, BEFORE analysing questions relating to Site, we should transform it into a Factor. Try typing:

```
> str(dat)
> dat$Site <- as.factor(dat$Site)
> str(dat)
```

As an interesting aside, now that we understand model formulation, we can now also use this format when we make our plots. Try these two plots:

```
> par(mfrow = c(1, 2)
> plot(dat$Species, dat$flush.dist)
> plot(flush.dist ~ Species, data = dat)
```

Simple post-hoc tests

In the previous example, we didn't need to run post-hoc tests because there was no difference in flushing distance among species. Let's work a slightly more complex example now.

Let's assume that the sites we sampled cover a range of levels of urbanisation so that Site 1 is most urbanised, followed by Site 2, then Site 4, then Site 3, with Site 5 being most rural. As you would expect, gulls are least affected by ORVs (although they take flight at the same distance as other birds, they land far closer to their take-off point than any other species - can you set and test hypotheses to show this?); our research question is whether urban gulls acclimate to ORVs (*i.e.*, whether rural gulls are more flighty than urban gulls).

Our hypotheses now look something like this:

H_0: Mean landing distance of gulls at all sites is the same
H_A: Mean landing distance of gulls at all sites is not the same

The first step is to isolate data only for gulls. You have done this before; do it again, then create a new model called `mod3` and generate an ANOVA table for your hypothesis test.

Note that the result is now highly significant ($p << 0.05$), indicating that there is indeed a difference in gulls' landing distance among Sites. To investigate further, we need a post-hoc test to identify where these differences are. Try typing:

```
> TukeyHSD(mod3)
> plot(TukeyHSD(mod3))
```

This provides a series of pair-wise Tukey tests (Honestly Significant Difference) indicating the difference in mean landing differences between pairs of sites (first column), the lower and upper confidence bounds for the difference (second and third columns, respectively) and the p-value associated with the hypothesis test that the difference is zero. The plot provides a visual representation of the table. Looking at these, we should notice that landing distances for gulls differ between all pairs of sites with the exception of Sites 2 and 4, which have among the highest levels of urbanisation.

Unfortunately, this representation of the Tukey test tells us something, but does not completely answer our question. To do that, perhaps we should plot the results of our test. To do this, we need a new package, so type:

```
> install.packages("gplots")
> library(gplots)
```

OR alternatively, click on the "Packages" tab of your "Files, Plots, Packages and Help" pane, and tick `gplots`; if not available, you can access it via the "Install Packages" tab of the Packages window.

Next try typing:

```
> plotmeans(land.dist ~ Site, data = gulldat)
```

This produces a neat(ish) graph of the mean landing distance by Site (± the 95% confidence interval). In general, if confidence intervals do not overlap, the difference between means is significant, so the plot closely reflects the results of the Tukey test, with landing distances shortest at Site 1, slightly greater at Sites 2 and 4 (which were indistinguishable), greater still at Site 3 and greatest at Site 5. This strongly suggests that rural birds are more flighty than urban birds, which may well have adapted their behaviour to accommodate frequent disturbances by humans.

General linear models and model selection

In the last two sections you have seen how to perform regression and ANOVA in R. You will have noticed that the way R handles them is similar: the formulae, and the `summary()` and `anova()` statements are the same, also. In fact, regression and ANOVA are the same and are united (along with ANCOVA, MANOVA and MANCOVA) under the banner of general linear models. General linear models assume a normal error structure and provide a general framework for including categorical and continuous explanatory variables in models.

The fundamental approach in statistics and in R is fitting models to data. The approach is to find the minimal adequate model. Here we will build a general linear model and the find the minimal adequate model.

Example: Zooplankton biomass and mine tailings

This example is part of a study of the potential effects of the Lihir gold mine (PNG), one of the largest gold mines in the world, on the marine ecosystem. The overburden and waste ore from the mine on Lihir island is released as a slurry into the ocean at a deep-sea tailing placement. This analysis is focused on whether the mine might negatively impact the biomass of zooplankton in the region, and is part of a much larger study focused on nekton, forage fish, and large pelagic fish, and in particularly, heavy metal biomagnification. The response is Zooplankton Biomass (dry weight.m^{-3}) and there is a mix of explanatory variables that are categorical – Zone (Inshore, Offshore), Region (Mine, Reference), Time (Day, Night) – and continuous – Depth (in metres) and Temperature (°C). The variable Barcode is a unique identifier for each sample, and for the purposes of our analyses can be ignored.

Load the data into R:

```
> Zooplankton <- read.table("LihirDW.csv", header = TRUE, sep = ",")
```

Note that `read.table()` is more general than `read.csv` as you can specify the separator (here a comma). Now, the first thing when you start an analysis it to get a feel for the data. Using the function head gives you the variable names and by default the first 6 rows of data (he we ask for more):

```
> head(Zooplankton, n = 20)
```

Finally, it is worthwhile checking that the class of the variables to make sure that R has interpreted them correctly. In R, categorical variables are of class factor, and continuous variables are of class numeric.

```
> str(Zooplankton)
```

Note that the levels of the class variables are ordered alphabetically – *i.e.*, Mine before Reference, and Inshore before Offshore. This is important to remember for the interpretation of results.

It is useful to look at the distribution of the data for each variable and the relationships between variables. We can do this with the pairs function:

```
> pairs(Zooplankton)
```

Remember that for each variable name along the horizontal, the variable is on the y-axis, and along the vertical, the variable is on the x-axis. A number of features are evident from this plot. First, there are lots of data points in horizontal or vertical lines, which is indicative of them being categorical (*i.e.*, they have several different but discrete levels). R plots them as integers starting at 1. You will see that Region has 2 levels, Zone has 2 levels and TimeOfDay has 2 levels. Second, Depth is not strictly continuous, but has been

sampled only at certain depths. Nevertheless, it is sufficiently continuous to treat it as a continuous variable. Last, we can start to see relationships that might exist in the data (but remember these are bivariate relationships only). Plots for Biomass on the y-axis (bottom row of the plot) suggest that there might be higher biomasses for the 1st levels of Zone (Inshore) and Region (Mine), and that it declines as Depth increases. There appears to be no relationship with Temperature. The continuous relationships do not look non-linear (*i.e.*, they do not look curved).

The initial model

It is easy to write the full model (*i.e.*, one with all terms) and plot it. For simplicity, we will fit a model without interaction terms. Here's how:

```
> model1 <- lm(Biomass ~ Depth + Temperature + Zone + Region + TimeOfDay,
data = Zooplankton)
> summary(model1)
```

Have a look at the output. Which variables are significant and which variables are not? Now to plot the model, we put 6 graphs on a page (2 rows by 3 columns) and then use `termplot()`:

```
> par(mfrow = c(2,3))
> termplot(model1, se = TRUE)
```

The `termplot()` function plots regression terms against the explanatory variables. Each term in the model has a separate plot (but note that all variables are in the model simultaneously). The interpretation of these plots is straightforward and is the relationship between the explanatory variable (on the x-axis) on the (partial) response (on the y-axis). Note that there has to be a partial effect for each explanatory term because this is an additive model and we get the predicted y-values by summing the effects of the different explanatory variables. Continuous terms are constrained so that the line of best fit goes through the mean of the explanatory variable and the mean of the (partial) response goes through 0 (as it does for categorical variables). There is a positive effect of the explanatory variable on the response when the response is above 0 and a negative effect of the explanatory variable on the response when it is below 0.

Here we set the `se` parameter to be true so standard errors are included, which are shown as dashed bands. For continuous variables, if a horizontal line can be placed between standard error bands for all terms, then that variable is not significant. What is your interpretation of the Depth and Temperature terms? For the categorical variables, when the standard-error intervals overlap then the levels are not significant. What is your interpretation of the Zone, Region and TimeOfDay terms?

Model diagnostics

How well behaved is this model in terms of homogeneity of variance and normality?:

```
> par(mfrow = c(2,2))
> plot(model1)
```

The residuals-vs-fitted plot provides a visual assessment of the homogeneity of variance assumption (best if there is little pattern to the residuals). Here we can see a tendency for the residuals to fan out for larger fitted values, which is quite common. A log transformation of the response could help here, as it makes the larger residuals relatively smaller. The Normal Q-Q plot also shows some deviation from normality (normally distributed data should be close to the line). A log transformation can often help with normality too. Let's see:

```
> model2 <- lm(log10(Biomass) ~ Depth + Temperature + Zone + Region +
TimeOfDay, data = Zooplankton)
```

```
> plot(model2)
```

Although the residuals are not perfect, they are better than using the raw response values, and the normality assumption is much improved. Let's stick with the log-transformed response, so model2 becomes our full model.

Now the next thing we need to do is to develop a procedure to be able to remove any variables that are not important. This is not as easy as it might sound…

Model selection

We are guided by the principle of parsimony, whereby we look for the simplest model that retains only significant variables, and we prefer simpler parameterisations (*e.g.*, linear terms rather than non-linear ones). Fit vs complexity. We will have a better fit with a more complex model, but it might not be very able to generalise.

There are a couple of issues here. The first is that because the terms in the model are often correlated, the significance of a particular term will be different, depending on what other terms are in the model. For example, we know that local temperature and not fungal growth will drive how many people go to the beach. However, fungal growth is positively related to temperature and thus indirectly to the number of people going to the beach. If we built a model with only fungal growth as a predictor, then it would probably be significant, but if we put the local temperature in as well, fungal growth probably would not be significant. We thus want to start (wherever possible) with all explanatory variables in the model.

Another major problem is that most designs (other than ANOVA experiments) are unbalanced. Balanced designs for categorical variables mean that each level of a factor has the same sample size. For unbalanced designs, the order of terms in the model is important (this is because R uses what are known as Type I sums of squares). The best protection against this problem is to work backwards from the full model (*i.e.*, starting with the model containing all available terms).

Now, how do we work backwards considering the order of terms can sometimes be important? A conservative approach is to use the `summary()` statement to identify the least significant variable and remove it, and then test whether removing the term reduces the predictive ability of the model (*i.e.*, it has explanatory power). This can be done by:

```
> summary(model2)
```

The least significant variable is Temperature (the Intercept must remain in the model). We can remove this using the `update()` function by changing the right side of the model formula:

```
> model3 <- update(model2, ~ . - Temperature)
```

The ~ represents we are dealing with the right side of the model formula, the `.` is the model as it stands, and `– Temperature` signifies we want to remove that term. Check that `model3` does not have Temperature:

```
> summary(model3)
```

Now let's see if Temperature was really not significant. We can use the `anova()` function, specifying two models, with the 2nd one nested within the first (*i.e.*, we are testing the difference in the two models – here Temperature):

```
> anova(model2, model3, test = "F")
```

Yes it is. Now TimeOfDay in `model3` is marginally significant ($p<0.1$) and we could choose to remove it or leave it in. For ecological problems where no one will die because of the outcome, it is OK to leave it in.

Why is it not best practice to remove multiple terms at once (that appear to be not significant)?

Fitting a model can be tricky, but the following approach is justifiable and repeatable and maximises your chance of identifying a good model:

1. **Fit the maximal model.** Put all the explanatory variables in the model
2. **Begin model simplification.** Inspect the parameter estimates and remove the least significant term (of those that are non-significant) first using `update`, starting with highest-order interactions (if you have specified interactions in your model; in our case, we did not)
3. If the deletion causes a **non-significant increase in residual variance** (known as deviance for generalised linear models) then **leave the term out** of the model
4. If the deletion causes a **significant increase in residual variance** (or deviance) then **return the term** to the model

Keep removing terms from the model, one at a time (*i.e.*, repeating Step 3 and 4), until the model retains only significant terms. We can automate this process of model selection using the `drop1()` function:

```
> drop1(model2, test = "F")
```

This performs an F-test comparing the overall model with the model resulting from removing that one specific variable per each line in the output table. It shows that removing the Temperature term is best and we could do this using the `update()` function. If we had dozens of variables in our model then `drop1()` can save time.

There is another method of model selection that many statisticians suggest is better: using Akaike's Information Criterion (AIC). AIC is a measure of the goodness of fit of the model. Models with more parameters fit better simply because the model itself contains more information. For example, if you had the same number of parameters as datapoints, the model has a perfect fit, but is complex (*e.g.*, a straight line with 2 parameters can always fit perfectly through 2 datapoints). In determining whether extra parameters are warranted in the model, the AIC considers the trade off between model simplicity and fit. It does this by penalising the inclusion of an extra parameter – it must reduce the residual sums of squares (deviance) by at least 2 to be retained. The model with the lower AIC is preferred.

We can compare the AIC models by using the:

```
> drop1()
```

The <none> model is the current one (*i.e.*, dropping no variables). The Depth variable clearly has the smallest AIC, but Temperature has an AIC that is larger than our current model and thus can be removed with `update()`.

Finally, the easiest way to use AIC and generate the final model automatically is to use `step()`:

```
> model4 <- step(model2)
> summary(model4)
```

Look through the output and see what it is doing. The function `step()` often retains variables that are only close to significant in the model. You could then conduct a manual analysis of the significance of the remaining variables.

Now let's plot the final model:

```
> par(mfrow = c(2,2))
> termplot(model4)
```

It is most useful with standard errors (set `se = TRUE`):

```
> termplot(model4, se = TRUE)
```

We can also plot the residuals (set `partial.resid = TRUE`):

```
> termplot(model4, se = TRUE, partial.resid = TRUE)
```

For a publication, we would want all lines to be black:

```
> termplot(model4, se = TRUE, partial.resid = TRUE, col.term = 'black',
col.se = 'black', col.res = 'black')
```

Finally, it is best to present the final fitted model and to list the non-significant terms (and to show the changes in residual sums of squares or deviance associated with each).

Interaction terms

In the above models, we considered only additive terms, and discovered that there were several variables contributing to the best-fitting model. Of these, the two most significant were Depth (continuous variable) and Zone (discrete variable). Remember that the term plot we did shows a single linear relationship between log10(Biomass) and Depth. But for argument's sake, lets say that we'd like to know if this relationship is really constant across Zones.

Your first response may be to plot the fits:

```
> with(Zooplankton, plot(log10(Biomass) ~ Depth, col = Zone))
```

Note that I have used the `with()` function to avoid repeatedly specifying the data frame; I have also used the model formula in the `plot()` function (for consistency with our modeling approach), and that I have allowed the colour of the points to vary by Zone. The result suggests that if there is a difference in the relationships, it is not large.

To formally model this situation, we need to include both main effects as well as an interaction term; we do this as follows:

```
> model5 <- lm(log10(Biomass) ~ Depth * Zone, dat = Zooplankton)
> summary(model5)
```

Note that we use the `*` to denote a full factorial model (all main effects, as well as interactions).

The model summary is very interesting. Bear in mind that the intercept refers to the reference case, which in this instance is `Zone = Inshore`. The first line of the summary output tells us what the intercept for the log10(Biomass)-Depth relationship is for the Inshore Zone, and that it differs significantly from zero. The second line gives the slope of the log10(Biomass)-Depth relationship for the Inshore Zone, and

indicates that this, too, differs significantly from zero. So the log10(Biomass)-Depth relationship for the Inshore Zone is significant. The third summary line tells us how much we need to add to the Inshore Zone's intercept to get the corresponding value for the Offshore Zone. This additional intercept differs significantly from zero, indicating that the intercepts for the Inshore and Offshore relationships are statistically distinguishable. Finally, the last line of the summary tells us how much we need to add to the slope of the log10(Biomass)-Depth relationship for the Inshore Zone to get the corresponding slope for the Offshore Zone. This is again significantly different from zero, confirming that both slope and intercepts for the two relationships differ.

In the ecological literature, this type of analysis is quite common, and is referred to as an analysis of covariance (ANCOVA). Note that this refers to the specific circumstance where one of the explanatory variables is continuous (the covariate) and the other is discrete. Of course, interactions aren't limited to this case: any mix of continuous and discrete variables is permitted. In each case, though, the interpretation is the same: if the interaction term is significant, the effect of one of the main factors depends on the level of another.

To confirm that the interaction term is significant in our example, we could type:

```
> anova(model5)
```

Alternatively, we could use our model-building techniques to determine whether a model without the interaction results has a significantly worse fit.

Finally, it is important to realize that we have all of the tools we need to output plots of such model fits. We start by getting the range of the Depths for Inshore samples:

```
> DepthI <- range(Zooplankton[which(Zooplankton$Zone == "Inshore"),]
$Depth)
```

There's a lot going on in this line. We'll work from the inside out. The `which()` function allows us to isolate rows of the Zooplankton data frame that contain Inshore Depths. Note that we are indexing rows with the `which()` function, so it is within square brackets and is followed by a comma to indicate that we want to return all columns. We are then simply requesting the `range` (minimum and maximum) of the variable Depth from this subsetted data frame. Do the same for the Offshore Zone yourself.

Next, we need to set up a data frame to accept predictions from the model:

```
> InDat <- data.frame(Depth = seq(DepthI[1], DepthI[2], 0.1), Zone =
rep("Inshore", length(seq(DepthI[1], DepthI[2], 0.1))))
```

This is fairly self-explanatory. The new data frame contains a variable called Depth, which itself contains a sequence of depths from the minimum to the maximum at intervals of 0.1 m. The second variable within the new data frame is called Zone and it simply repeats the word Inshore for each cell. It is no coincidence that the variables in this data set have names identical to the terms in the model. Construct a similar line of code yourself for the Offshore Zone.

Next, we need to predict values from the model for each row of our new data frame, and add these values to the data frame:

```
> InDat$Fit <- predict(model5, newdata = InDat)
```

Here we have simply asked R to predict values of `model5` for each combination of variables in `InDat`. Now get R to do this for the Offshore Zone.

Finally, adjust the plotting parameters (so that we're plotting only one graph in the plot window), plot the original data, and then add the fitted line:

```
> par(mfrow = c(1, 1))
> with(Zooplankton, plot(log10(Biomass) ~ Depth, col = Zone))
> with(InDat, lines(Fit ~ Depth, col = "black"))
```

When you have added your own code to plot the fit for the Offshore Zone, you will notice how different the relationships really are.

Generalised linear models

The linear models we dealt with in the previous section assume a normal error structure, a response that is both continuous and theoretically unbounded (*i.e.*, can include large negative and positive values). Many kinds of statistical problem violate these assumptions:

1. Count data (the response is an integer and often contains many zeroes), for which the error structure is commonly Poisson (a discrete distribution)
2. Count data expressed as proportions, for which the error structure is usually binomial
3. Binary response variable (*e.g.*, male or female; success or failure), for which the error structure is commonly binomial

Generalised linear models encompass linear models with normal error structure as well as those with Poisson and binomial structure. Luckily, running generalised linear models is straightforward. It is usually as simple as specifying the family for the error structure (*i.e.*, poisson for count data and binomial for proportions and binary responses).

```
> glm(y ~ x, family = poisson)
```

or even simpler:

```
> glm(y ~ x, poisson)
```

Below is a summary of some of the different types of situations that you can apply generalised linear models to. It is worthwhile discussing these in some detail.

Properties	**Continuous**	**Count**	**Proportions**	**Binary**
Examples	Abundance Density	Frequencies Number of animals Number of occurrences	Sex ratios Proportion responding Infection rates Percentage mortality	Dead or alive Male or female Healthy or diseased Occupied or empty Mature or immature
Error structure	Normal	Poisson	Binomial	Binomial
Errors	Variance constant	Variance increases with mean. Data whole, positive numbers	Variance maximum @ p=0.5	Variance maximum @ p=0.5
Response	All real numbers	Integer > 0	Between 0 and 1	Between 0 and 1
Formula (Resid dev < Resid df)	glm(y~x,normal) **or** glm(y~x)	glm(y~x,poisson)	y~cbind(#success,#fail) glm(y~x,binomial)	glm(y~x,binomial)
Model selection (Resid dev < Resid df)	anova(m1,m2, test="F")	anova(m1,m2,test= "Chi")	anova(m1,m2,test= "Chi")	anova(m1,m2,test= "Chi")
Formula (Resid dev > Resid df)	NA	glm(y~x,quasipoisson)	glm(y~x,quasibinomial)	NA
Model selection (Resid dev > Resid df) – **Note: different tests**	anova(m1,m2, test="F")	anova(m1,m2,test="F")	anova(m1,m2,test= "F")	anova(m1,m2,test= "Chi")
Default link	Identity	Log	Logit (ln(p/q))	Logit

Inverse link	Identity	Exp	Inverse logit	Inverse logit

Note that the `anova()` test to compare two models is changed from using an "F" to a "Chi" statistic for non-normal families (actually many people might say their family is non-normal!). Further, when the residual deviance is larger than the residual degrees of freedom (the ratio of these is the error), then the model fits poorly and the "quasi" form of the family is specified. This fits an extra parameter to account for unexplained variance. It also changes the anova tests of two models from "Chi" to "F". For more information on using the quasibinomial and quasipoisson parameters, see Zuur (2007) on Analysing Ecological Data.

It is also necessary to know a bit about the three components of a generalised linear model: the error structure, linear predictor and link function. A summary of these is below.

Component	**Explanation**	**Why?**
Error structure	Allows the specification of different error structures	Errors are non-normal (strongly skewed, kurtotic, strictly bounded, such as proportions, or cannot be negative, such as counts)
Linear predictor	The right hand side of the model equation glm(y~x+z); for each prediction of the model it is the linear sum of the terms for each of the parameters	
Link function	This function relates the predicted y-values to the linear predictor. The inverse link transforms the values from the linear predictor to the predicted y-values	Keeps values bounded. Predicted counts must be positive and predicted probabilities between 0 and 1

Example: A binary response

For the next Intergovernmental Panel of Climate Change (IPCC) report, we have performed a meta-analysis of the global literature to find out whether changes in marine biological time series >20 years in length are consistent or not with climate change. The world map in the Simple Graphics section shows the data coded by red (not consistent) and blue (consistent). The response is Consistency (coded as 0 = not consistent or 1 = consistent) and is thus a binary response. Explanatory variables are Taxa (10 biological groups), Latitude (tropical, subtropical, temperate, polar), and Obstype (Abundance, Calcification, Community change, Demography, Distribution, Phenology). The key question is whether the proportion of observations consistent with expectations under climate change is affected by Taxa, Latitude or Obstype.

Import the file `ConsistencyData.csv`. Start by confirming the variable names and their types. To get a feel for the data, start by using `tapply()` to look at the mean and number of samples for Consistency for each explanatory variable. Note that if climate change did not affect marine life, then you would expect 50% of the data being consistent and 50% not consistent with climate change. What do the results mean values from the `tapply()` suggest?

Build the full model with Consistency as the response and Taxa, Latitude and Obstype as predictors, with a binomial error structure. Use a model-building approach to come up with the best model. Use `termplot()` to plot the final model. Note that to see the x-labels on the graphs you will need to tilt them vertically (HINT: consider the `las` parameter).

What do the results tell you?

Challenge question

Let's suppose that we want to use the model we just derived to understand climate-change responses for a typical temperate marine food chain containing zooplankton, fish and seabirds. And suppose that we are interested only in the most common response types: abundance, distribution shifts, and phenological shifts. We might anticipate that the shortest-lived critters would respond most strongly and consistently, but that as you travel up the food web, ecological interactions over the lifespan of the organism could result in lower levels of consistency and greater variability in response.

From previous sections, we know that we will need a prediction matrix. We start by specifying the levels of the variables of interest:

```
> tax <- c("Zooplankton", "Bony fish", "Seabirds") # Focus taxa
> lat <- c("Temperate") # Set the value of Latitude to temperate
> obs <- c("Abundance", "Distribution", "Phenology") # Climate responses
```

Next, we use the function `expand.grid()` to generate a prediction data frame (see help for more information) containing all combinations of the specified variables:

```
> ndat <- expand.grid(Obstype = obs, Taxa = tax, Latitude = lat) # Make
a data frame of all combinations of predictors
```

We then simply make the predictions:

```
> preds <- data.frame(predict(model1, newdata = ndat, type = "link", se
= TRUE)) # Do the prediction
```

Note that we have specified the `type` of prediction as `"link"`, which means that we predict in terms of the GLM model, in other words as log-odds ratios. We can also request the standard error in the same units. Log-odds ratios are slightly unintuitive, but not that difficult to grasp. The odds ratio is simply the ratio of the probability of success (the observation is consistent with climate change) relative to failure (the observation is not consistent with climate change). Taking the natural logarithm of this value gives you the log-odds ratio.

The next step is to build a data frame that can be used to plot. Start by binding together the two data frames we have:

```
> predat <- cbind(ndat, preds) # Combine the data
```

Next, calculate the asymptotic confidence bounds for the estimates (the estimate ± 1.96 × the standard error of the estimate):

```
predat$LCL <- predat$fit - (1.96 * (predat$se.fit)) # Lower bound of asymptotic 95% CI
predat$UCL <- predat$fit + (1.96 * (predat$se.fit)) # Upper bound of asymptotic 95% CI
```

Now we can back-transform everything. Remembering that the odds ratio is:

$$\frac{p(success)}{p(failure)} = \frac{p(success)}{(1 - p(success)}\,,$$

the back-transform to p(success) can be shown to be:

$$p(success) = \frac{\exp\,(log{-}odds\ ratio)}{(1{+}\exp\,(log{-}odds\ ratio))}\,.$$

So, back-transforming everything to proportions (= p(success)) is easy:

```
> predat$cons <- exp(predat$fit)/(1 + exp(predat$fit)) # Back-transform
the predictor
> predat$lcl <- exp(predat$LCL)/(1 + exp(predat$LCL)) # Back-transform
the LCL
> predat$ucl <- exp(predat$UCL)/(1 + exp(predat$UCL)) # Back-transform
the UCL
```

Now, we make the plot. Start by making sure that we have a single plot in the window:

```
> par(mfrow = c(1, 1)) # A 1 x 1 matrix of plots
```

Next, set up an empty plot of appropriate size (we're plotting the three trophic levels on the x-axis, so we need it to run from 1 to 3, with a bit of space on either side, so we can spread out estimates for the different response types for each taxon). Note that we use `xaxt = "n"` to avoid plotting the x-axis ticks and labels:

```
> plot(c(0.5, 3.5), c(0, 1), type = "n", xaxt = "n", xlab = "Taxon",
ylab = "Proportion consistent with climate change (± 95% CI)", font.lab =
2, las = 2) # Make an empty plot with appropriate axis labels
```

Then, we add the names of the taxa on the x axis, at the appropriate points:

```
> axis(1, at = 1:3, label = levels(predat$Taxa)) # Add new x axis
```

We add a horizontal line at 0.5 to indicate the expectation under chance; *i.e.*, where p(success) = p(failure), so the odds ratio is 1 (and the log-odds ratio is 0 – useful for model fitting, not so?)

```
> abline(h = 0.5, lty = 3)
```

Finally, we add the points and confidence intervals. Note that because R overplots existing objects at each step, start with lines for the confidence intervals, and only afterwards add points. Note also, that this is done by brute force, specifying each confidence interval separately, then adding coloured points. Later, when you have learned about for loops, you can do this more elegantly. Alternatively, if you grub around in Rseek for a few moments, you will find that there are numerous packages that contain ready-to-use functions for plotting error bars. For the time being, though, it probably helps if you understand how it is done from first principles. Finally note that although Zooplankton is labeled at 1, Fish at 2 and Seabirds at 3, x values are specified symmetrically around these points; this simply achieves the desired aim of spreading out the estimated points so that they are easily seen:

```
>  lines(rep(0.9, 2), c(predat$lcl[1], predat$ucl[1])) # CI for
Abundance
> lines(rep(1, 2), c(predat$lcl[2], predat$ucl[2])) # CI for
Distribution
> lines(rep(1.1, 2), c(predat$lcl[3], predat$ucl[3])) # CI for Phenology
> points(c(0.9, 1, 1.1), predat$cons[1:3], pch = 21, col = c("blue",
"gold", "red"), bg = c("blue", "gold", "red"), cex = 2) # Points for
Abundance, Distribution and Phenology
> lines(rep(1.9, 2), c(predat$lcl[4], predat$ucl[4])) # CI for Abundance
> lines(rep(2, 2), c(predat$lcl[5], predat$ucl[5])) # CI for
Distribution
> lines(rep(2.1, 2), c(predat$lcl[6], predat$ucl[6])) # CI for Phenology
```

```
> points(c(1.9, 2, 2.1), predat$cons[4:6], pch = 21, col = c("blue",
"gold", "red"), bg = c("blue", "gold", "red"), cex = 2) # Points for
Abundance, Distribution and Phenology
> lines(rep(2.9, 2), c(predat$lcl[7], predat$ucl[7])) # CI for Abundance
> lines(rep(3, 2), c(predat$lcl[8], predat$ucl[8])) # CI for
Distribution
> lines(rep(3.1, 2), c(predat$lcl[9], predat$ucl[9])) # CI for Phenology
> points(c(2.9, 3, 3.1), predat$cons[7:9], pch = 21, col = c("blue",
"gold", "red"), bg = c("blue", "gold", "red"), cex = 2) # Points for
Abundance, Distribution and Phenology
```

Finally, using another new trick, add a legend. Note that the position of the upper-left corner of the legend box is determined in graph units and specified using arguments `x` and `y`. Next, provide a list of plotting symbols and their colours (note that the `bg` argument from `points()` becomes `pt.bg` here, to differentiate the intent from filling the legend box with a colour). Finally, provide a list of labels for the coloured symbols, in order. Simple.

```
> legend(x = 0.4, y = 0.18, pch = rep(21, 3), col = c("blue", "gold",
"red"), pt.bg = c("blue", "gold", "red"), legend =
levels(predat$Obstype)
```

Homework: Day 2

Using the data from the Homework exercise on Day 1, address the following research questions:

1. Is there a relationship between driftline ammonium concentrations and degree of urbanisation? If so, what does it look like?
2. Is there a difference in the amount of ammonium oxidised among beaches? If so, where are these differences?
3. Does the degree of urbanisation impact the degree of nitrification (ammonium oxidisation) on beaches?
4. Assume that instead of having paired samples from the driftline and effluent line, these samples were drawn at random. Using the model-building techniques provided, and a glm, construct the best possible model explaining variation in ammonium on the beaches of the Sunshine Coast.

R Introductory Workshop

DAY 3

Multivariate statistics, Part I (cluster analysis)

What are multivariate statistics?

So far, we have been dealing with univariate statistics; in other words, statistics pertaining to only a single response variable. What happens when we are interested in many response variables simultaneously? The answer is simple, we move to a thing called multivariate statistics. These sound scary, but in reality, they are not.

Understanding the concept

Perhaps before starting with multivariate analysis proper, let's start with something we're familiar with: geography.

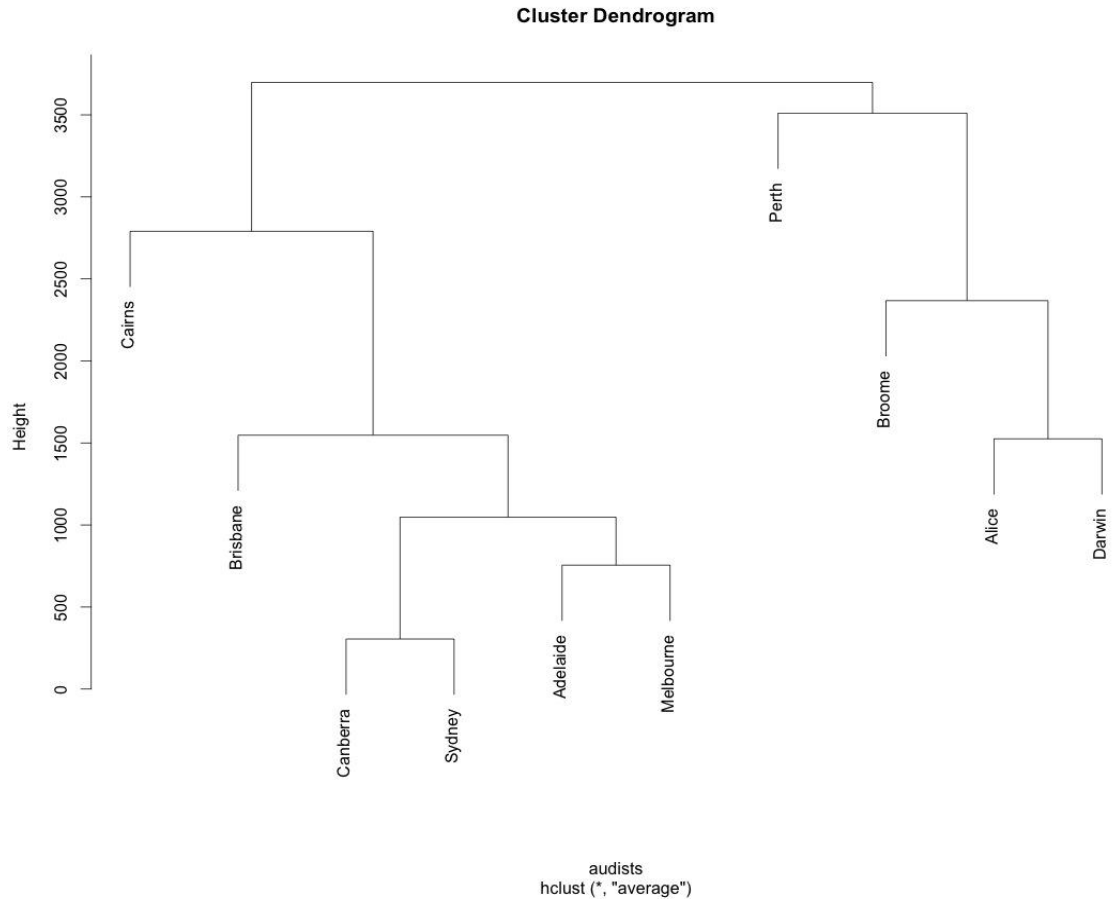
A simple way of thinking about the positioning of the cities marked on this map (two dimensions) is in terms of driving distances (one dimension): when distances between cities are small, the cities are close together, and when they are large, the cities are far apart. This much is obvious.

Here's a table (from the R package `DAAG`) of driving distances (in km) between the cities plotted:

	Adelaide	Alice	Brisbane	Broome	Cairns	Canberra	Darwin	Melbourne	Perth
Alice	1690								
Brisbane	2130	3060							
Broome	4035	2770	4320						
Cairns	2865	2415	1840	4125					
Canberra	1210	2755	1295	5100	3140				
Darwin	3215	1525	3495	1965	2795	4230			
Melbourne	755	2435	1735	4780	3235	655	3960		
Perth	2750	3770	4390	2415	6015	3815	4345	3495	
Sydney	1430	2930	1030	4885	2870	305	4060	895	3990

Inspecting this table, we can see that Sydney, Canberra, Melbourne and Adelaide are close to each other, whereas Perth is a long way from just about anywhere. In this way, we have gone from a visual representation of the distances in two dimensions (the map) to a numerical representation in one dimension (the table).

A different way of representing these numbers visually is called a dendrogram (quite literally a "tree drawing"), and it looks something like this:



In this branching format, cities close together in space are arranged close together on the same branch. So, in this dendrogram, Canberra and Sydney are closest together, with Adelaide and Melbourne close by;

Brisbane is closer than most other cities, but not that close; and Cairns is further away, still. Cities to the north and west form a completely separate branch. Because we are looking for "clustering" of objects on the dendrogram, the process by which we derive the dendrogram is unsurprisingly called clustering.

We won't worry ourselves with the technicalities of the process here, but let's convince ourselves that R can replicate the analysis. Start by accessing the distance matrix in `DAAG`. It is called `audists`. Try typing:

```
> install.packages("DAAG")
> library(DAAG)
> data(audists)
> audists
```

This makes the data matrix available and displays it. Why is only half of the matrix provided?

To plot the dendrogram, type:

```
> plot(hclust(audists, method = "average"))
```

The results should be exactly the same as those provided above. In this case, `audists` is called the distance (or resemblance) matrix, and provides a measure of how "dissimilar" (in this case, close) sites are to each other (a small dissimilarity means that distance is small and cities are close together; by convention, similarity = 1 - dissimilarity). The argument `method = "average"` is just informing R to use what is known as group-averaged clustering. There are other types, but this is the most common in ecology, so we present it here.

A real-world example

There are some data in a file called Sharks.xlsx. These data are annual catches of different types of sharks on drumlines from ten sites along the Australian east coast. Import these data to a data frame called `sharkdat`, and then have a quick look to see what they look like.

Distance (resemblance) measures

In univariate analyses, we compare samples on the basis of univariate summary statistics (can you remember how to add up the number of sharks caught at each site, or to calculate the average number of sharks caught per site?). Such univariate measures, however, are obviously useless when analysing multivariate data. Instead, we need to figure out a way to compare each sample with every other sample, just like we did with distances in the example above. Can you think of a measure that correlates patterns among samples?

Does the function `pairs()` ring any bells? If it does, you will remember that it plotted a matrix of correlations among samples (columns) in a data frame. We can't use the function directly here because our data not only have samples as rows, but they also include additional explanatory variables (State and Site). What do we do now?

First, extract the data that we want. Try typing:

```
> sharks <- sharkdat[,3:7]
> sharks
```

With this, we ask R to assign the third to seventh columns of the data frame `sharkdat` to a new data frame called `sharks`. We then print sharks to the screen to have a look at the data. When we do this, we notice that columns are still species, with sites as rows, so we need to flip the data. Try typing:

```
> sharks <- t(sharks)
> sharks
```

The call to the function `t()` transposes the data frame (*i.e.*, rows become columns and columns become rows), which is what we want.

Now try pairs:

```
> pairs(sharks)
```

This plots the number of each species of shark for each site against the corresponding number in every other site. Where the plots fall on more or less a straight line going from lower left in the panel to upper right, there is a strong similarity in pattern. We called this a correlation before. Just as we used a call to `cor.test()` before to test the correlation between two variables, we can use a call to function `cor()` to calculate the correlation coefficient between the pairs of samples in the plotted matrix. Try it:

```
> cor(sharks)
```

This gives us a matrix of correlation coefficients that are close to 1 where the pair plot has a close-to-straight line (*e.g.*, 0.996 for var 5 vs var 6; *i.e.*, Site 5 vs Site 6). Where the lines are more messy, as is the case var 7 and var 9, the correlation is smaller (0.795).

Let's try using this as a measure of distance (resemblance) among samples in a cluster analysis. First, make the correlation matrix into a lower-diagonal "distance" object. This sounds complicated, but the call is simple and logical, and a "distance" object is simply an object in the form of the distance matrix we had for Australian cities. REMEMBER, though, that a small distance means a close relationship (samples close to one another), but large correlations mean a close relationship. In other words, whereas we want our distance (resemblance) matrix to contain dissimilarities, it currently contains similarities. To make correlation work, we could therefore simply transform the similarities into dissimilarities by the simple formula provided earlier. To do this, type:

```
> corshark <- as.dist(1-cor(sharks))
```

Now, lets see what the dendrogram looks like:

```
> plot(hclust(corshark, method = "average"))
```

If you view the original data frame again, you will notice that sites 1, 3, 4, 9 & 10 are from Queensland; the rest are from New South Wales. Does the dendrogram reflect this geographic pattern?

Generalising the approach

The correlation coefficient worked alright for this example, but it wouldn't work well in all cases, partially because it models a linear relationship between variables, and we don't always need this to be the case. A more generally-applicable resemblance measure for ecological data is the Bray-Curtis similarity index. We don't need to know exactly what this looks like (luckily we can leave this to the computer, although it's not as complicated as you might expect). The advantages of the Bray-Curtis index are that for any pair of samples:

- It is 100 when samples are identical (all species are present at both sites at the same abundance).
- It is 0 when samples have no species in common.
- It does not vary with scale of measurement (provided that all samples are measured on the same scale).

- It is unaffected by the addition of a species that occurs in neither of the two samples.
- It is unaffected by the addition of a new sample to the matrix.
- It registers differences when species are present at the same proportions, but different abundances.

Bray-Curtis's main drawback is that its value can be dominated by species with very large abundances. We'll explain how to deal with this later, though.

For the meantime, let's apply the Bray-Curtis index to our cluster analysis. There is just one slight complication: while the calls to `pairs()` and `corr()` require samples to be columns, all formal multivariate analyses require samples to be rows. We can correct this easily enough. Type:

```
> sharks <- t(sharks)
> library(vegan)
> BCshark <- vegdist(sharks, method = "bray")
> BCshark
```

We can see from the screen print that we again have a lower triangular matrix representing "distances" as dissimilarity (*i.e.*, high values indicate great dissimilarity, so corresponding samples would be placed far apart on a dendrogram). Plot the dendrogram to see:

```
> plot(hclust(BCshark, method = "average"))
```

Although this is a fairly simple example, it clearly shows the power of clustering as a technique for exploring structure in multivariate data sets.

Besides the group-averaged, hierarchical agglomerative clustering technique that we have applied here, there are many other techniques available in R. We won't deal with these here, but the approaches to analysis are fairly similar in each case.

Multivariate statistics, Part II (nMDS)

Background

Clustering is a good start to multivariate analysis, but it's certainly not the most powerful approach. Perhaps while we were discussing visual representations of clustering, you were wondering why we prefer an essentially one-dimensional dendrogram to a two-dimensional map. The real answer, of course, is that we don't. Visual representations in two dimensions are almost always better than those in one, simply because they convey at least twice as much information. Once we get beyond two dimensions, though, we start to trade off information content against simplicity of interpretation. So the question we now face is how we derive two-dimensional representations using multivariate statistics?

The answer is non-metric multidimensional scaling (nMDS). This sounds complicated by comparison with clustering, but in reality it isn't. All nMDS does is employ a "brute force" or "bucket and spade" approach to fitting data. The routine is simple: the computer repeatedly throws the sample points onto a two-dimensional space (or three-, or more dimensions, depending on the user's specification) and then checks to see how well the distances between the sample points on this so-called "ordination" reflect the dissimilarities in the distance (resemblance) matrix. Each new ordination is a slight modification of the previous one. Each time the fit improves, that distribution becomes the working solution and the process is repeated. When a certain number of random redistributions of the samples in the ordination space fails to improve the "fit", the solution is considered "optimal".

NOTE that highly similar points need to be close together to indicate close relationship, so distance reflects dissimilarity (= 1 - similarity). NOTE also that this is non-metric, so we're more interested in the rank order of dissimilarities than in their absolute values.

Let's assume that we're back to the geographic distribution of Australian cities and the computer's first (random) attempt produces a plot that looks like this:

This isn't a great representation of the relative distances held in the distance matrix. But remember that the computer keeps throwing out rearrangements of the samples until it can no longer improve the fit.

This sort of approach is a bit like a hillwalker in the fog - if s/he needs to find lower ground s/he would simply walk downhill and carry on doing so until stepping in any direction means going uphill again. But s/he can't see the full landscape, so could become trapped in a local valley. nMDS minimises this possibility by restarting the routine MANY times, each time with a different arrangement!

Of course in reality, there is a trick because the computer STARTS with a metric scaling fit that is close to the theoretical best possible (which is likely NOT the best, just close to it).

The measure of how well the ordination distances match the resemblances is called the STRESS (which in R ranges between 0 and 1). The smaller the Stress is, the better.

Applying the technique:

Taking this knowledge, let's try to do an nMDS on the distance data. Remember that in this context, we're using road distances as measures of dissimilarity (so that large distances indicate very dissimilar positions on a map). Given that we have the distance matrix already, try typing:

```
> library(vegan)
> mdsAus <- metaMDS(audists)
> plot(mdsAus)
```

This outputs a bunch of points in two-dimensional space. From the theory we discussed, we should know that this is the optimal arrangement of cities in two-dimensional space according to the road distances among each pair of cities. But a plot of points is relatively unhelpful. What can we do?

Try inspecting the `metaMDS` object by typing:

```
> names(mdsAus)
```

This outputs all of the different bits and pieces of data associated with the `metaMDS` object (you can do the same thing with any other object in R; try it, you may find it useful). You can play around with the contents of any of these, but the one we're really interested in is the points (the things we just plotted). So try typing:

```
> mdsAus$points
```

This outputs the names of the cities along with their coordinates in the nMDS ordination space. When we plotted the `metaMDS` object earlier, we plotted the coordinates without lables. Try replotting the ordination, but adding labels.

```
> plot(mdsAus)
> text(mdsAus, pos = 2, offset = 1)
```

The call to `plot()` prints the points, the call to `text()` prints the names associated with those points (these are all the data we saw in the matrix produced by `mdsAus$points`). The `pos` option tells R whether to plot the text below (1), to the left (2), above (3) or to the right of (4) the point; whereas the `offset` option tells R how far away from the point, these labels should be printed. This should all be easy enough to understand.

Does the resultant nMDS plot look anything like the original map? Well, it should, sort of, just rotated clockwise through about 90º. Generally in nMDS biplots, the arrangement in space is arbitrary, so there is no need to rotate the plot, but in this case, check the fit to the map using rotation. To do this, tell R how to rearrange the plot. Generally this is done, by feeding R some environmental variables. In this particular case, we can cheat a little and tell R what the longitude of each city is. To do this, try typing:

```
> degE <- c(138.5833, 133.8700, 153.0333, 122.2360, 145.7797, 149.1314,
130.8333, 144.9667, 115.8585, 151.2086)
> plot(MDSrotate(mdsAus, degE))
```

Logically enough, the call to `MDSrotate()` simply rotates the `metaMDS` object along the axis defined by the specified environmental variable. Here, supply an ordered list (the order is determined by row names in `mdsAus$points`) of longitudes for the cities. NOTE that this does not add any element of longitude to the output plot, but merely rotates the points so that the environmental variable (longitude) runs from left to right along the x-axis. Add the city names (this time half a line above the points):

```
> text(MDSrotate(mdsAus, degE), pos = 3, offset = 0.5)
```

We now have a plot that looks pretty much like the map, which shows just how good the nMDS routine really is. There is one minor issue, though: Perth is plotted far to the south of its actual position. Why? What would the plot look like if you used direct distances between cities rather than road distances?

Back to an ecological problem:

Import the drumline catch data for sharks that we used previously to a data frame called `sharkdat`. Next, select only data pertaining to catches themselves, and write these data to a data frame called `sharks` (we've done this before, so it is not repeated here). Now, let's repeat our cluster analysis using nMDS.

```
> mdsShark <- metaMDS(sharks, distance = "bray", autotransform = FALSE)
```

Note that for this routine, we can work with the raw catches, and specify that `metaMDS()` derives the distance (Bray-Curtis dissimilarity in this case) matrix internally. We also need to specify a second option, telling `metaMDS()` not to transform our data automatically, but rather to work with raw counts. In many instances, transformation may be necessary, but not here (more on this at the end of the section). Plotting the output is interesting:

```
> plot(mdsShark)
```

There are two little clusters of circles (samples) and five red crosses (species). This tells us not only how similar samples are to each other, but also which species contribute to the splits between groups.

Let's clarify the output by inspecting the `metaMDS` object, and subsequently adding information to the plot:

```
> names(mdsShark)
```

We're interested in two matrices:

```
> mdsShark$points
```

which gives us the coordinates of the samples (circles) within the ordination space; and:

```
> mdsShark$species
```

which gives the coordinates of the species in the ordination.

We could add the site numbers to this plot, but it would be more interesting to see if the sites cluster according to State. So let's try the following:

```
> plot(mdsShark)
> text(mdsShark$points, labels = sharkdat$State, pos = 3, offset = 1)
```

So here the nMDS biplot is plotted, then one line above each ordination point a label is plotted, determined by the name in the variable `sharkdat$State`, which is, of course, the State of origin (not in the rugby league sense!) for the corresponding sample. The output shows that the Queensland shark catches are very different from New South Wales shark catches.

In this case, we can inspect the data to see where these differences lie (NSW has more Grey Nurses, Great Whites and Bronze Whalers, whereas Queensland has more Tiger and maybe Bull sharks). Can we plot these? Of course we can:

```
  > text(mdsShark$species, labels = row.names(mdsShark$species), pos = 3,
offset = 1)
```

These plots were done from first principles, but they can also be done directly through the `metaMDS` object. Try typing:

```
> plot(mdsShark, type = "n")
> points(mdsShark, display = "sites")
> text(mdsShark, display = "spec", pos = 3, offset = 1)
```

Here, we have simply used the display option to tell R what elements of the ordination we want to plot.

An analogy to understand what an nMDS biplot represents

Our output so far has been easy to understand, but multivariate analyses are seldom so clear. Because nMDS represents multi-dimensional space in two or three dimensions, things can get pretty messy. But that doesn't mean that results are difficult to understand: we are all familiar with the compression of three dimensions into two-dimensional space. Think about a shadow. This is a two-dimensional representation of three dimensional space. BUT, not all shadows really give us much of a hint about the sort of thing they are caused by. Depending on the orientation of the object relative to the light source, the shadow can transmit very little information, or quite a bit. So, nMDS essentially tries to create a "shadow" of your multivariate data that conveys most information (*i.e.*, it reflects as truly as possible patterns in those data).

Using environmental data to explain ordinations

When we were plotting the map, we reorientated it using longitude as an environmental variable. Here, we need not reorientate the nMDS biplot, because it is intuitive enough, but we may wish to ask the question of whether State is a significant predictor of the community structure in shark catches. To do this, we construct a model, as we have done before. Try:

```
> statefit <- envfit(mdsShark ~ sharkdat$State)
```

Here, we are using the `envfit()` function, which determines the centroid of the points in an ordination (here `mdsShark`) associated with a discrete variable (factor), in this case, `sharkdat$State`. It also assesses the significance of its explanatory power on the basis of randomisation tests. The results are easily accessible:

```
> statefit
```

and these results can be added to the ordination plot by typing:

```
> plot(statefit)
```

Here, the results are exactly as you would anticipate. Although analyses can get a LOT more complex, `envfit()` is quite good at modelling multiple discrete (factors) and continuous (vectors) variables simultaneously.

When to transform ecological data

One aspect of multivariate analysis that we mentioned before, but didn't resolve, is data transformation. The Bray-Curtis similarity index is notorious for becoming very large (indicating large dissimilarities) as soon as there is a large disparity in abundance of even one species between a pair of sites. When we were analysing shark catches, this wasn't much of an issue, because the sharks were about the same size, and are probably equally important to answering the question that we were interested in (are there patterns of community structure in shark catches along the Australian east coast?). In many other situations, we may be interested in variability in numbers of both abundant and rare species. For example, in a rocky shore survey, there may be thousands more barnacles than chitons, and hundreds more mussels than sea squirts. In such cases, to avoid the multivariate analysis being overwhelmed by signals from the barnacles, we would consider transforming the data before starting the analysis. If the difference in numbers is only one order of magnitude, we may go with a square root transformation (this has little effect on small numbers, but makes large numbers substantially smaller). If, however, there are two or more orders of magnitude difference in the abundance (or whatever other measure you're working with) for the "species" you're interested in, you may want to go with a logarithmic or fourth-root transform.

Note that for cluster analysis, you need to force this transformation manually, but metaMDS() has the ability to inspect your data and automatically select an appropriate transformation. In our example above, we switched this ability off. Try switching it on and see what happens. How can you find out what transformation has been employed?

Whatever the case, be sure that you take the transformation into consideration when interpreting your analysis. Failing this, you could potentially come up with some rather silly statements.

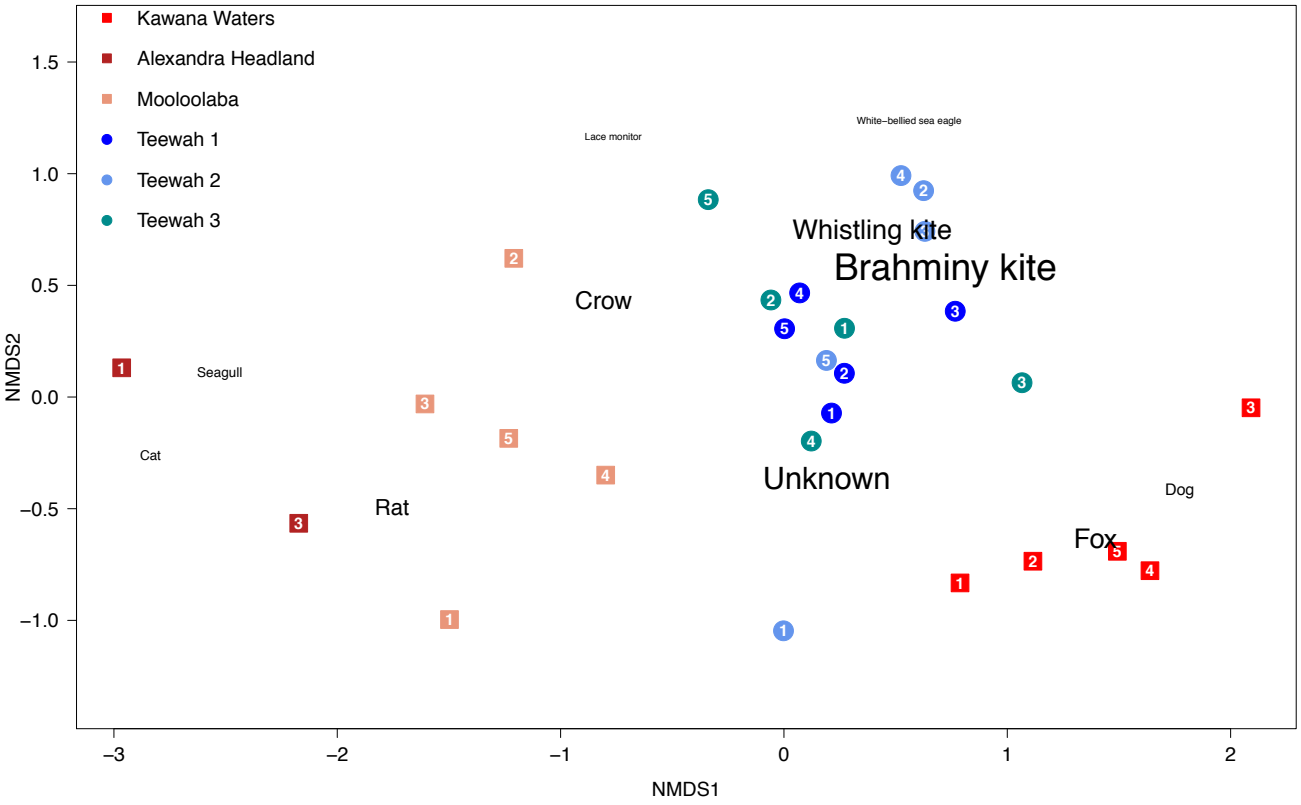
Other ordination techniques

R of course has the ability to run MANY other types of multivariate analyses, including (but by no means limited to) significance tests on cluster analyses (package `pvclust`), unsupervised partitioning (`kmeans()` in the default `stats` package, and `pamk()` in package `fpc`), PCA (`princomp()` in the default `stats` package), ANOSIM (`anosim()` within the `vegan` package) and even PERMANOVA (`adonis()` within the `vegan` package). None of these techniques is any more difficult than those we have discussed here.

Why use R when there is a perfectly good alternative (PRIMER)?

Over the past two decades, many ecologists, and especially marine ecologists have come to know and love PRIMER for multivariate analyses. Without wanting to discourage support for the good folk at PML who have brought us this excellent piece of software, I now personally default to R for my multivariate analyses. I do this not only because I run a Mac (so do not have native access to PRIMER), but also because I find the flexibility of analysis and graphics in R hugely beneficial. For example, throughout my days as a PRIMER user, I had to export graphical outputs for subsequent editing in a vector graphics package. But with a few lines of R code, I can produce the finished plot in one fell swoop. To illustrate, I have included

below an nMDS biplot of beach scavenger community structure (Huijbers et al. *in prep*) for urban (squares in shades of red) and non-urban (circles in shades of blue) beaches along the Sunshine Coast. Replicate trials across sites are identified by superimposed white numerals, and the influence of individual scavengers on the ordination is indicated with text size scaled to the fourth root of frequency of occurrence. Not only can I insert this image directly into our manuscript, but having developed the code, I can save it so that I can quickly and easily adjust the analysis if required, I can recycle large parts of it in future analyses, and I can share the code with my co-authors so that they can easily check my working, or modify the code for their own purposes.



Having said this, the package `vegan` currently lacks some aspects of the functionality of PRIMER. However, ongoing development is adding new routines constantly, and additional inspiration from non-PRIMER analyses are also broadening `vegan`'s horizons.

Extra exercise

The file called Molluscs.xlsx contains the data in Table 5 of Harrison MA and Smith SDA (2012) Cross-shelf variation in the structure of molluscan assemblages on shallow, rocky reefs in subtropical, eastern Australia. *Marine Biodiversity* **42:** 203-216. Try to analyse these data to explore the hypotheses that structure mollusc communities on benthic reefs varies across the continental shelf. HINT: If you think about the structural requirements of data submitted to multivariate analyses in R before importing the data, you may save yourself some time.

Programming basics

The art of programming is to lay out sequential sets of instructions (script/code) in your Source Editor that is easily understood by both R itself, and also by a naïve reader. Before we get too far into this section, here are some (biased – from my own personal perspective) tips on good (as opposed to best) practice in simple coding for R:

- Annotate your code liberally. Use the hash character (#) to indicate comments – anything appearing on a line subsequent to a # will not be executed by R, so use entire lines to provide headers for sections of code, and also feel free to add comments at the end of lines of code (everything before the # will execute as usual). Annotations will remind you of what you have done and why; they will also guide the naïve reader when you share your code.
- Indent lines (using TABs or SPACEs) to indicate which bits of code are subsets of the broader script.
- Use spaces in your code to prevent ambiguities between for example `a <- 3` and `a < -3`.
- Wrap long lines of code so that they are easily viewed in a moderately size source editor window. R anticipates that lines ending in a comma or an opening bracket are wrapped.

Bearing these suggestions in mind, let's illustrate some principles of programming by writing a short program to construct a correlation matrix for the shark catch data we used previously.

Start by opening a new Source Editor window, and providing a few annotated comments to explain what you're doing. Next, import the shark data (you should have made a .csv of it before, but if not, the original data are available in the file Sharks.xlsx). Given that we want to work only with the columns that contain catch data (and not the descriptive variables for State or Site), subset the data frame to only retain variables of interest.

The next step is very important when programming in R (or for that matter, most other languages): assign an object to collect results, and at the same time, allocate sufficient memory to that object. This is called preallocation. While this is not important for a small data set like the one we are working on, it can be vital to prevent variables from "expanding" sequentially when analyzing particularly large data sets. This significantly slows processing time, often exponentially. Here we create a new matrix object as follows:

```
> cormat <- matrix(rep(-99999, times = ncol(sharkdat)^2), ncol =
ncol(sharkdat), byrow = TRUE)
```

The function `matrix()` specifies the type of object. Because we want the catch of each shark to correlate with the catch of other sharks, as well as itself, we need a square matrix with as many rows and columns as there are columns of shark catch data; there are therefore `ncol(sharkdat)^2` cells in the matrix. For each cell, we use the replicate or `rep()` function to generate a large negative number (bear in mind that correlations are from -1 to 1, so a value of -99999 would really stand out in the matrix) `ncol(sharkdat)^2` times. This is called initialisation. We specify the number of columns for the matrix with `ncol = ncol(sharkdat)`, and then specify that the `-99999`s are to be read into this matrix by row (the order doesn't really matter in this context, but can be important in other contexts). We are now set to catch the results of an analysis that will have 25 steps (the number of cells in the matrix).

To undertake these 25 steps, we need a function that will repeat a calculation 25 times. This is where a for loop is very handy (as it is anytime we need to iterate a process). In R, for loops are specified with the function `for()`, which accepts two simple arguments: a variable that acts as a counter (it takes on a single value at a time) and a sequence of numbers (which are the values passed to the counter). Immediately following the `for()` call, is an expression (set of instructions) enclosed in curly brackets "`}`".

In our example, we want to repeat a calculation for each column in the data frame of shark catches, so our sequence takes the form `1:ncol(sharkdat)`; the counter can be specified as any valid variable name, but convention means that the letter `i` is most common:

```
> for(i in 1:ncol(sharkdat)){}
```

The expression inside the curly brackets should now be the focus of our attention. Everything else that we write in this loop constitutes that expression, so goes between the curly brackets!

Bearing in mind that the formula for the Pearson correlation coefficient is:

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

we need to specify a vector of residuals for column `i` of the data frame. This is simply done (if have included the `for()` function to reiterate the structure of the loop):

```
> for(i in 1:ncol(sharkdat)){
>   x <- sharkdat[,i] - mean(sharkdat[,i])
>   # Everything else in for loop goes here and in subsequent lines
> }
```

Note that I wrapped the line after opening the curly bracket, and again before closing it. Note also that I have indented lines to ensure that I know where the for loop starts and stops.

Next, we need to work on `y`. Since this is merely the same formulation as `x` simply applied to another variable, specifying `y` should be easy. BUT remember that for each time we specify `x`, we need to specify `y` 5 times. In other words, we need a second (nested) for loop:

```
> for(i in 1:ncol(sharkdat)){
>   x <- sharkdat[,i] - mean(sharkdat[,i])
>     for(j in 1:ncol(sharkdat)){
>       # Everything else in for loop goes here and in subsequent lines
>     }
> }
```

Note how I have again indented lines so that the structure of the for loop remains obvious. In this case, the expression within the second (internally nested) set of curly brackets will calculate `y` (and then will compute and write the results):

```
> y <- sharkdat[,j] - mean(sharkdat[,j])
```

Now that we have both `x` and `y`, we can use them to calculate the correlation coefficient according to the formula above (remember that this is to be done 25 times, so this remains within the second (internally nested) set of curly brackets):

```
> r <- sum(x * y)/(sqrt(sum(x^2)) * sqrt(sum(y^2)))
```

The last two lines to be added inside the second set of curly brackets will assign the correlation coefficient to the correct place in the matrix that we set aside to collect results, first for cells on the diagonal or above:

```
> cormat[i, j] <- r
```

and then for cells on the diagonal or below:

```
> cormat[j, i] <- r
```

Your final for loop should look like this:

```
> for(i in 1:ncol(sharkdat)){
>  x <- sharkdat[,i] - mean(sharkdat[,i])
>    for(j in 1:ncol(sharkdat)){
>      y <- sharkdat[,j] - mean(sharkdat[,j])
>      r <- sum(x * y)/(sqrt(sum(x^2)) * sqrt(sum(y^2)))
>      cormat[i, j] <- r
>      cormat[j, i] <- r
>      }
>  }
```

You can now highlight the entire for loop and run it. All being well, your correlation matrix `cormat` should provide the same results (rounding notwithstanding) as the call:

```
> cor(sharkdat)
```

Now let's assume that we wanted to identify the sharks responsible for the strongest correlation. We could add row and column names to the matrix and hunt visually, as follows:

```
> rownames(cormat) <- colnames(cormat) <- names(sharkdat)
> cormat
```

Here, the functions `colnames()` and `rownames()` are fairly self explanatory; as we know, `names()` returns the names of the variables in the data frame `sharkdat`.

Instead of doing the search manually, we could write a line or two of code to do the search for us. We can access the strongest correlation by typing:

```
> max(abs(cormat))
```

but of course, because all the values on the diagonal are 1, the result here is obvious. To get where we need to go, we will need to modify the code in our previous script. Try replacing the lines reading:

```
> cormat[i, j] <- r
> cormat[j, i] <- r
```

with an `if()` statement. These are fairly straightforward accepting only a conditional argument (*i.e.*, an argument that sets a condition). The function is followed by an expression that is to be implemented if the expression is true. Where necessary, this is followed in turn by `else` and a second expression that is to be implemented if the condition is false. Remember that expressions are enclosed within curly brackets.

In the situation we have, we want to replace values on the diagonal of our matrix with an `NA`. It is easy to evaluate when we have a result that fits on the diagonal: for these cases, `i = j`. This gives us a hint as to how we might set up our `if()` statement:

```
> if(i != j){} else {}
```

This reads, if `i` is not equal to `j`, then do whatever is in the first set of curly brackets, or else do whatever is in the second set of curly brackets. All that remains is to figure out what these expressions are. Of course, for off-diagonal values, we want the correlation coefficient returned, so simply place the two lines of code doing this between these brackets. The second expression will assign r on the diagonal a value of NA. The script will look something like this (remember to break long lines at commas or brackets):

```
> if(i != j){
>    cormat[i, j] <- r
>    cormat[j, i] <- r
>       } else {
>       cormat[i, j] <- NA # Place an NA on the diagonal
>            }
```

Once you have inserted this code, your request for a maximum correlation should return a value, provided you remember to account for NAs. However, although you have now identified the strongest correlation, you still have to look it up manually, or do you? Try typing:

```
> grep(max(abs(cormat), na.rm = TRUE), cormat)
```

The function `grep()` searches for a `pattern` (the first argument) within a `vector`. In this case our `pattern` is the maximum absolute value of an off-diagonal correlation coefficient, and our `vector` is the matrix of correlation coefficients read as a single array of values by row. The function returns two values because the correlation coefficients are reflected above and below the diagonal. We could use either of them to identify the sharks, but lets pick the first one, just for simplicity:

```
> ccell <- grep(max(abs(cormat), na.rm = TRUE), cormat)[1]
```

We can now figure out which row of the matrix is involved as:

```
>    rrow    <-    ceiling(grep(max(abs(cormat),    na.rm    =    TRUE),
cormat)[1]/ncol(cormat))
```

Once we have done that, finding the column is as easy as:

```
> ccol <- ccell - ((rrow - 1) * ncol(cormat))
```

From there, we can identify the sharks involved as:

```
> names(sharkdat)[c(rrow, ccol)]
```

While the example used here was fairly trivial, this exercise will have demonstrated how useful simple programming structures can be when querying data frames, especially where iterative procedures are required; it should also have demonstrated how straightforward the logic of programming is.

Intermediate programming: functions

Function basics

Most tasks in R use functions built in the R base package (*e.g.*, `mean()` or `sum()`), or are built into external packages (*e.g.*, `glm()`). A function applies code to the input arguments and returns an object. For instance, the function `mean()` sums a vector (a series of numbers) and divides by the number of elements.

We can also define our own functions in R. Writing your own functions is useful for organizing code and for re-applying the same task to multiple different data sets. Further, many functions in R take a function as an input and apply this input function to a table or list of numbers (*e.g.*, `tapply()`)

Defining a function is simple. Functions must be defined by the program before they are used, for instance they can be defined at the start of a script or in another script that is called separately.

Function names follow the same rules as variable names (they can't start with numerals, they are case sensitive, and they should have no special characters).

First, lets calculate the standard error of some data.

```
> x = c(5, 1, -3, 5, 7, 9, -2) #data
> n = 7 #number of samples
> sd(x)/sqrt(n) #standard error
```

We can easily take the line for code for calculating the standard error wrap it up in a simple function that calculates the standard error from a vector of data `x`.

```
> stnderr <- function(x, n){sd(x)/sqrt(n)}
```

This code tells R that `stnderr` is a function. Inputs to the function (called arguments) are contained in the brackets. In this case there are two inputs `x`, the vector of values, and `n` the sample size. The code contained within the curly braces `{}` is run with the input values. The value `sd(x)/sqrt(n)` will be returned by the function.

If we have data in a vector called `tail.length` (in the FishLengths.csv data set; import this to a data frame called `dat`), then to calculate the standard error and assign it to a new variable we can use our function:

```
> setail <- stnderr(dat$tail.length, length(dat$tail.length)
> setail
```

As you can see, we used variables `x` and `n` when we defined the function. Now if we replace the name `x` with any vector, and `n` with the number of tail length values, R will apply our function to that data. R will use the values in the order they are entered, so make sure that the inputs are used in the function in the same order they are defined. Alternatively, we can also tell R which values belongs where by specifying the original variable names:

```
> stnderr(n = length(dat$tail.length), x = dat$tail.length)
```

In fact, we can avoid this potential order problem by defining the sample size within the function, using the `length()` function applied to `x`:

```
> stnderr <- function(x){
>  x <- na.omit(x)
>  n <- length(x)
>  sd(x)/sqrt(n)
>  }
```

Try typing:

```
> stnderr(c(10.1, NA, 12.3, 15.1, 8.9, 8.1, 10.0))
```

Now type in:

```
> n
```

Does it give you the value of `length(n)`? No. This illustrates that variables assigned within functions are local to that function only. This means that `n` does not exist outside the function. Further, the function will only return the last unassigned value (in this case sd(x)/sqrt(n)).

Functions in other functions

Functions like this are useful if we want to perform the same task multiple times. We could use this function inside the R function `tapply()` to calculate the standard error of groups in a data frame. If we have a data frame with two variables (here `tail.length` is the response and `fishsp` is the explanatory variable) and we want to know the standard error of the tail length for different kinds of fish, then we can use our function:

```
> with(dat, tapply(tail.length, fishsp, stnderr))
```

and `tapply()` will apply our function to tail lengths grouped by fish species.

An exercise to try on your own

Now that you know how to do a little programming, and you can write your own functions, try to find a way to use the stnderr function and a for loop to plot means with error bars for the last example in the ANOVA section (we used the function `plotmeans()` before – can we do any better on our own?).

Intermediate programming: Applications for graphics

R can be used for mapping and GIS applications. There is a range of packages that cover applications including map projections, building rasters, spatial models, GIS overlays, more complex GIS routines and even google maps in R.

We will start with the basics and plot a map of seagrass distribution in Australia. Seagrass data are from the UNEP seagrass database (Green EP, Short FT (2003) World Atlas of Seagrasses. University of California Press, Berkley, USA) and the maps are in the R package 'maps'.

First, we need to load some packages, set the working directory and load the data:

```
> library(maps)
> setwd("myfolder")
> SG <- read.csv("seagrassaus.csv", header= TRUE)
> attach(SG)
```

Note that until now, we have avoided attaching data frames. What this does is allow direct access to variables without having to first name the data frame; this saves some typing (*i.e.*, we can avoid typing `datafram$` each time we refer to a variable).

Now let's look at the data:

```
> names(SG)
> nrow(SG)
```

What seagrass families are in Australia and how many?

```
> sg.fam <- levels(family)
> sg.fam
> nfams <- length(sg.fam)
> nfams
```

Let's start the mapping by plotting the coordinates of all the seagrass locations and then overlay with a map of Australia, using the maps package:

```
> plot(lon, lat, ylab = "Latitude", xlab = "Longitude")
> map(database = "world", xlim = c(100, 180), ylim = c(-60, 0), add=
TRUE)
```

We have used `xlim` and `ylim` to define the plotted region (just Australasia).

Next, let's make a map where each seagrass point is coloured according to family.
First we define a vector of colour names for each family in the database, using the base function `rainbow()`.

```
> famcols <- rainbow(nfams, start = 0, end = 10/12)
```

`rainbow()` takes as its arguments the number of colours, and the start and end points on a rainbow sequence (numbers between 0 and 1).

Now the plotting. First, set up a plot frame with axes. The argument `type = "n"` within `plot()` stops the plotting of the points. We will plot them later with different colours.

```
> plot(lon, lat, ylab = "Latitude", xlab = "Longitude", type = "n")
```

Then put the map of Australia on:

```
> map(database = "world", xlim = c(100, 180), ylim = c(-60, 0), add=
TRUE, fill= TRUE, col = "grey")
```

Now we will use points() to plot the seagrass locations. We will do this in a `for` loop, stepping through each seagrass family. The function `which()` is used to get indices for the rows of the database that correspond to just one seagrass family at a time (indices are stored in the variable `thisfam`). In essence, this subsets the data by identifying which elements of a dataset return the value `TRUE`. The `col` argument is used to specify the colour of the points. The call `famcols[i]` selects just one colour from our list of colours for each family.

```
> for (i in 1:nfams){
>  thisfam <- which(family == sg.fam[i])
>  points(lon [thisfam], lat[thisfam], col = famcols[i], pch=20)
>  }
```

Finally, we can add a legend, so we now what the colours represent. The functoin `legend()` takes a location (in this case the bottom left), names for the legend (seagrass families) and the corresponding point types and their colours. We need to be sure that the symbol colours in the legend and legend names are in the same order as they were used in the plotting above, otherwise the legend will be wrong.

```
> legend('bottomleft', legend = sg.fam, pch = 20, col = famcols)
```

If you have finished these tasks and there is still time, why don't you try plotting a map where each genus has a different symbol and then saving the map as a pdf.

For more advanced mapping and GIS tools check out the packages `raster`, `mapproj`, `googleVis`, `rgdal` and `sp`.

Extra study in programming: a randomisation test using a resampling function

Now that we have established some basic principles of programming, we can progress to more advanced applications. An alternative to the standard hypothesis-testing approaches of parametric and non-parametric statistics is randomisation tests. This approach has far fewer assumptions and is thus more generally appropriate. For example, a simple and intuitive way to test for a significant difference between the mean of two groups is to use a randomisation test. Say we want to know if the tail length of fish species A is significantly greater than the lengths for fish species B. We first calculate the difference in the means from the data, this is our test statistic. Then we randomly re-assign fish species names to the data and recalculate the difference in mean tail lengths 1000s of times, this gives us a distribution of test statistics. If the test statistic (the value we observed in our actual sample) is greater than 95% of the values from the randomisations, then we have significance at the 5% level.

First, let's generate our own data, so we know the real answer:

```
> N = 50 #Number of fish overall
> fishA <- rnorm(N/2, mean = 20200, sd = 1500) # random length values
for species A
> fishB <- rnorm(N/2, mean = 20000, sd = 1500) # random length values
for species B
```

These lines of code simply generate two vectors, each containing 25 random numbers from a Normal distribution. One of these distributions has a mean of 2200 and the other with a mean of 2000; both have a standard deviation of 1500.

```
> # make a factor
> fishnames <- factor(c(rep('A', N/2), rep('B', N/2)))
```

This line makes a factor (discrete variable) containing 25 "A"s, followed by 25 "B"s.

```
> tail.lengths <- c(fishA, fishB)
```

Here, we compile the random measures generated previously into a single vector by appending `fishB` to `fishA`.

```
> # calculate means
> fishmeans <- tapply(tail.lengths, fishnames, mean)
> fishmeans
```

Given our knowledge of `tapply()`, we know that this code simply calculates the mean tail length for each fish (A and B). The object `fishmeans` will therefore be an array containing two values.

```
> # calculate test statistic (one tailed value, we would use absolute
difference for a two-tailed test)
> teststat <- fishmeans [1] - fishmeans [2]
```

The test statistic is simply the difference in the means for the two species. The aim of our randomisation test will be to see how many ways a value at least as extreme as this may have resulted from the two samples we have to hand. We achieve this by simply shuffling the species identity of each tail length randomly.

```
# write a function that takes the data frame and randomly reorders the
labels and returns the test statistic
> randord <- function(x, xnames, N){
>  i <- sample(1 : N, size = N) # randomly re-ordered indices
>  xmeans <- tapply(x, xnames[i], mean) # calculate means, reordering
data randomly
>  xmeans[1] - xmeans[2] #test statistic
>  }
```

There are two tricks in the function that we may not have seen before. The first is `sample()`, which takes a sample of size `size` a vector at random, without replacement. As we have specified the vector to be `1:N` and the `size` to be `N`, we are simply asking R to randomly shuffle the numbers 1 to `N`. The next little trick is to order the array of species names according to this random order using `xnames[i]`, where `xnames` is the list of names supplied to the function and `i` is the randomly shuffled list. In essence, `i` indexes the order of the names so that they are random.

```
> # Now let's apply the function randord() nperm times and store the
test statistic value
> nperm <- 1000
> testvals <- rep(-99999, nperm) # preallocate a vector filled at the
start with -99999s
```

As before, it is just good practice to pre-allocate an object (and some memory) to catch the results. The subsequent for loop is self explanatory.

```
> for (i in 1:nperm){
>  testvals[i] <- randord(tail.lengths, fishnames, N) # apply our
function in a loop
>  }

> # look at the distribution of test statistics
> hist(testvals, 20)
>  points(teststat, 0, col = "red") #puts a red circle where the
estimated value lies
```

The function `hist()` used here simply takes a vector of values and constructs a frequency histogram. This allows us to inspect the outputs in a straightforward way.

```
> # calculate p-value
> sum(testvals > teststat) / nperm
```

We end by calculating up the proportion of random test values that are larger than the test statistic (as calculated on the basis of the original dataset). If this value is smaller than 0.05, we can assume that our test statistic is unlikely to have arisen by chance. Note, however, that if you run the routine repeatedly (without generating new tail lengths each time), you will get slightly different results. This is not an error, but simply a feature of randomisation tests, which rely on random reorganisation of data rather than on formal statistical distributions.

For more details see page 44 of the R user manual at http://www.r-project.org/

Some final thoughts

Hopefully, this brief introduction to the functionality of R will have whetted your appetite to learn more. The initial learning curve is steep, but you should more or less have overcome that over the past three days. The good news is that the R community is extremely friendly and welcoming. Questions are entertained with enthusiasm rather than antagonism, and sharing is encouraged at all times. We hope that you take this philosophy to heart and establish supportive R mini-communities in your labs.

Below is a list of broader resources that we consult on a regular basis. We hope that you will find them useful, and that you will let us know if you find other useful websites and the like.

Name	Target	www
Quick-R	A great guide for new R users	http://www.statmethods.net/index.html
RSeek	THE search engine for R solutions	http://www.rseek.org
R Reference card	Everybody – the top 100 or so R functions	http://cran.r-project.org/doc/contrib/Short-refcard.pdf
R Manual	Beginner to intermediate users	http://cran.r-project.org/doc/manuals/R-intro.html
R Base package	Summary of R Base functions	http://www.math.montana.edu/Rweb/Rhelp/00Index.html
FAQs	Everybody – frequently-asked questions	http://cran.r-project.org/doc/FAQ/R-FAQ.html
LURN	Beginners	http://r-resources.massey.ac.nz/lurn/front.html
simpleR	Beginners	http://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf
R Fundamentals	Beginner to intermediate (good for manipulating data)	http://faculty.washington.edu/tlumley/Rcourse/R-fundamentals.pdf
Idre, UCLA	Everybody - great list of general resources	http://www.ats.ucla.edu/stat/r/
Revolution analytics	Everybody - great list of general resources	http://www.revolutionanalytics.com/what-is-open-source-r/r-resources.php
R graphics gallery	Everybody – good code resource for graphics	http://gallery.r-enthusiasts.com/

Beyond this, the general R community can be accessed via the R Mailing Lists (http://www.r-project.org/mail.html). Please check though the list to see which may apply to you, and sign up. Please do read the FAQs and posting guides before submitting questions. Although people are generally friendly, even when you violate the protocols, it really isn't good form to waste other peoples' time.